USENIX

ANAHEIM CONFERENCE PROCEEDINGS

SUMMER

1990

USENIX

CONFERENCE PROCEEDINGS

June 11-15, 1990
Anaheim, CA

USENIX Association

Proceedings of the Summer 1990 USENIX Conference

June 11 – June 15, 1990
Anaheim, California, USA

For additional copies of these proceedings contact
USENIX Association
2560 Ninth St., Suite 215
Berkeley, CA 94710 USA

The price is $22.
Outside the U.S. and Canada please add
$15 per copy for postage (via air printed matter)

Past USENIX Technical Conferences

1990 Winter	Washington, DC
1989 Summer	Baltimore
1989 Winter	San Diego
1988 Summer	San Francisco
1988 Winter	Dallas
1987 Summer	Phoenix
1987 Winter	Washington, DC
1986 Summer	Atlanta
1986 Winter	Denver
1985 Summer	Portland
1985 Winter	Dallas
1984 Summer	Salt Lake City
1984 Winter	Washington, DC
1983 Summer	Toronto
1983 Winter	San Diego

TABLE OF CONTENTS

PLENARY SESSION

Wednesday (9:00-10:30) Chair: John Mashey

SHARED LIBRARIES AND DYNAMIC LINKING

Wednesday (11:00-12:30) Chair: Heinz Lycklama

DISTRIBUTED SYSTEMS

Wednesday (2:00-3:30) Chair: Joe Moran

OPERATING SYSTEMS I

Wednesday (4:00-5:30) Chair: Lori Grob

CONCURRENT SESSIONS TALK

Thursday (9:00-10:30)

Special session on computer generated music — how do computers make music, and how are they being used in music production, arrangements, and compostition.

Peter Langston, Bell Communications Research; Mike Hawley, MIT Media Lab; Eedie and Eddie, (201) 644-2332

OPERATING SYSTEMS II

Thursday (11:00-12:30) **Chair: Tom Ferrin**

FILE SYSTEMS I (Parallel Session)

Thursday (2:00-3:30) **Chair: Doug Comer**

APPLICATIONS (Parallel Session)

Thursday (2:00-3:30) **Chair: Doug Kingston**

LANGUAGES

Thursday (4:00-5:30) **Chair: Larry Rosler**

LESSONS LEARNED

Friday (9:00-10:30) **Chair: Clem Cole**

PERFORMANCE

Friday (11:00-12:30) Chair: Pat Parseghian

WINDOWING

Friday (2:00-3:30) Chair: Jim Gettys

FILE SYSTEMS II

Friday (4:00-5:30) Chair: Bill Shannon

PREFACE

Welcome to the Summer 1990 USENIX Technical Conference and Exhibition. The papers here were chosen after evaluating the 111 extended abstracts that were submitted for the Program Committee's consideration.

This conference's informal theme is "Beyond Mere Data: Perspective, Insight, and Understanding." We asked authors to tell us not just what they had done, but to emphasize the lessons learned from their work, to make comparisons with others' work, to describe tradeoffs. In general, we're hoping that you'll see some critical thinking about where we are, how we got here, and why.

In line with this theme, we've been fortunate not only to receive many interesting papers, but we've been lucky enough to get Dennis Ritchie to give us the Keynote talk on "What Happens When Your Kid Turns 21," which I expect should not only have Dennis' usual high level of insight, but some amusing surprises as well.

The technical sessions have a few other unusual features as well. In some cases, the talks will be a little shorter than usual, allowing time for the speakers to form a panel for more discussion and interaction with the audience.

As another unusual feature, you'll notice that one of the "Concurrent Sessions" talks occupies the first time slot on Thursday morning. We expect this to be a talk with especially broad interest, so we avoided placing another technical session directly opposite it.

Prefaces should be brief, and this one concludes thanks to the numerous people who have contributed to this effort: it is more work than I would have thought, so I'm happy to have had many knowledgeable people working on it. These include many reviewers who helped the Program Committee, and Judy DesHarnais and Ellie Young for continual help and advice. Finally, Evi Nemeth once again has managed the difficult task of putting together the Proceedings, with the help of several others.

John Mashey
Program Chair

TECHNICAL PROGRAM COMMITTEE

Clem Cole
Cole Computer Consulting

Doug Comer
Purdue University

Thomas Ferrin
Univ of CA at San Francisco

James Gettys
Digital Equipment Corp.

Lori S. Grob
Chorus Systemes

Douglas P. Kingston III
Morgan Stanley & Co.

Heinz Lycklama
Interactive Systems Corp.

John Mashey, *Chair*
MIPS Computer Systems

M. Douglas McIlroy
AT&T Bell Laboratories

Joe Moran
Lagato Systems, Inc.

Pat Parseghian
Princeton University

Lawrence Rosler
Hewlett-Packard Company

Bill Shannon
Sun Microsystems,Inc.

CONFERENCE ORGANIZERS

John Mashey, *Technical Program Chair*
(MIPS Computer Systems)
Judith F. Desharnais, *Meeting Planner*
(USENIX Association)
John Donnelly, *Tutorial Coordinator*
(USENIX Association)

PROCEEDINGS TYPESETTING

Rob Kolstad, *Proceedings Typesetting*
(Sun Microsystems, Inc.)
Evi Nemeth, Dotty Foerst, Darren Hardy, Trent Hein, Paul Kooros, Yasantha Samarasekera, *Proceedings Production*
(University of Colorado, Boulder)

AUTHOR INDEX

USENIX Summer Conference
June 11-15, 1990
Anaheim, California

James Q. Arnold – AT&T Bell Laboratories

ELF: An Object File to Mitigate Mischievous Misoneism

ABSTRACT

An object file is much like a shipping box: People typically ignore the container and focus on the contents. Despite its unobtrusiveness, a system's object file directly affects services such as program execution and development, and a deficient file format restricts the services a system can deliver. Moreover, some issues surface only as the computing environment changes. For example, an object file with mixed or ambiguous byte order complicates file sharing across a heterogeneous network, without causing problems for a homogeneous environment.

ELF (Executable and Linking Format) is the object file for System V Release 4. Unlike previous object files, ELF includes an access library that provides key services such as host/target translation and version control. In combination, the new object file format and the access library eliminate many problems and support new services.

Besides the technical aspects of object file design, this paper addresses the practical problem of changing object file formats. Object files contain the binary representations of programs, in which people may have substantial investments of time or money. Introducing ELF was made more challenging by the need to preserve investments and to limit the turmoil for people who write programs, for people who port System V to other architectures, and for people who use applications.

INTRODUCTION

Object files hold binary program images. Variations exist, but many systems support two main kinds of object files. First, compilation systems generate *relocatable* object files that correspond to input source code. Second, a program called the link editor combines relocatable files to create *executable* object files. One can run the program in an executable file because it is a complete image. On the other hand, relocatable files are partial images and typically are not suitable for execution. Because many services associated with program execution or development rely on the object file format, its design affects the efficiency, portability, and convenience of the operating system, compilation systems, and other programs. Depending on the target processor and operating system release, various object files have been used on the UNIX[1] system. Summaries of three typical designs follow.

The venerable *a.out* format was designed for the PDP-11.[2] It provided 8-character symbol names, a maximum of 5,461 symbols per file, separate sections for instructions (called text) and initialized data, and relocation entries for every text and data word. The format was elegantly simple but inappropriate for larger address spaces.

For 32-bit machines, the *a.out* format was extended in several ways. Most obviously, 16-bit quantities were enlarged to 32-bit values. The symbol table changed to allow names of unlimited length. Relocation entries also changed significantly. Larger programs and different relocation conventions made it necessary to associate a relocation entry with an explicit address, instead of relying on the implicit correspondence between program sections and relocation records. Because many systems with various object file formats use *a.out* as the generic executable file name, I'll avoid confusion by using the term "classic *a.out*" for the format described here.

Another format, called the Common Object File Format (COFF), was designed primarily to support electronic switching systems (the telephone network). Its distinguishing features were multiple sections (text, data, uninitialized memory, reserved memory, overlays, etc.), some support for multiple target processors, defined structures for symbol tables and relocations, and debugging information tailored for the C language. Eventually, COFF became the object file for System V.

Over the last few years, these formats' deficiencies have become increasingly apparent. Nonetheless, the thought of designing, implementing, and introducing another format daunted us. Our

[1] UNIX is a registered trademark of AT&T.

[2] PDP is a registered trademark of Digital Equipment Corporation.

internal costs concerned us, as did the indirect costs on other computer vendors and software developers. Despite some promising experimental object file designs, those efforts stayed out of System V.

As a major release, System V Release 4 contained many new features, including 4.3BSD compatibility. Not only were the System V and BSD object files[3] different, but some new features compromised object file compatibility in subtle (or not so subtle) ways. Moreover, we wanted to support various RISC processors, and neither COFF nor the BSD format was up to the job. We also wanted to use SVR4 as the platform for the *System V Application Binary Interface* (ABI), enabling binary application portability [1]. Dynamic linking and a capable object file were keys to this effort, and the existing formats were unsatisfactory. Having tangible benefits at hand and knowing we could not guarantee compatibility for all COFF files, the step to a new format seemed reasonable. After deciding to change object files, we explicitly established the following requirements on the new format.

- *Support multiple processors.* The UNIX system pioneered operating system portability [2]. Each machine could have its own object file (and many did), but a common format makes the operating system and compilation systems more portable.
- *Make cross development convenient.* Cross development systems let one compile and build a program on one machine, while the execution will occur on a dissimilar processor. Many organizations use various machines, connected with networks and distributed file systems. This heterogeneity encourages a mixture of native and cross development, with object files being shared as needed. An object file should be designed to accommodate cross development (including byte order differences) to take full advantage of distribution.
- *Allow efficient, flexible program loading.* Program loading (the *exec* system calls) and paging consume a significant portion of many systems' resources. If the object file makes *exec* particularly efficient, system throughput improves. We also wanted to support nonmonolithic programs with facilities such as dynamic linking and shared libraries. Letting multiple processes share common code can decrease memory usage and improve software maintenance.
- *Adapt to new requirements over time.* In its most basic use, an object file merely transmits program binary code to the processor. Often, though, it assists more elaborate facilities, such as program debugging, configuration management, program patching, etc. Prior experience taught us that we could not foresee everything the object file might eventually have to do. Consequently, we wanted the new format to be adaptable.
- *Tolerate multiple programming languages.* Traditionally, most programs on the UNIX system have been written in C. Nevertheless, compilers for other languages exist, and the object file format must support those programs. Particularly in the area of symbolic debugging information, previous object files have inadequately supported non-C languages. Although we wanted good C support, we wanted to avoid C chauvinism in the design.
- *Scale to larger or smaller architectures.* All immediate targets were 32-bit machines, but we wanted to make it simple to support other classes of machines. For example, 64-bit machines exist today, and they need the same basic services from an object file. In other words, we wanted an object file family, the first member of which would fit 32-bit machines.

I mentioned the System V ABI above, and a few words of explanation are in order. The System V Interface Definition, also known as the SVID, defines *source* interfaces [3]. By so doing, it describes how to write programs that will compile and run on any system that supports the SVID interfaces, independent of the system architecture. The ABI serves a similar purpose but focuses on *binary* interfaces. Because those binary interfaces depend, in part, on the target processor, the ABI comes in two parts. The System V ABI applies to all implementations; companion processor supplements define characteristics that apply to individual architectures. Thus if several computer companies elect to follow the ABI for a particular processor, an application developer can build an application once and have it run on all those vendors' machines. This decreases the developer's effort and makes more applications available for the vendors' systems.

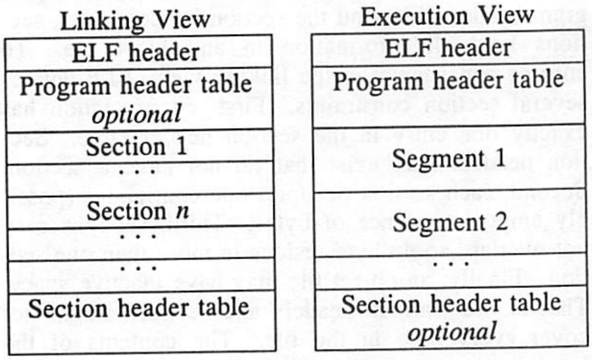
Following sections describe the important characteristics of the file format and the accompanying access library. With this groundwork established, I then examine specific design decisions and relate them to achieving the goals listed above. Some items may appear to conflict at first glance, such as efficiency and flexibility. While possibly true in theory, several techniques helped reduce the conflicts.

[3] The BSD object file is based on the classic *a.out* format.

FILE FORMAT

As I mentioned before, program execution and program development are the primary activities that manipulate object files. Secondary activities occur as well, but we specifically wanted to streamline the normal cases. If necessary, we were willing to accept minor performance penalties or inconveniences for secondary activities to support program execution and development.

Normally, the output of a compilation is a relocatable object file that corresponds to the input source file. The link editor combines an application's relocatable files and system libraries to build the executable program. Relocatable and executable files do not necessarily have the same constraints, and we considered using two file formats. Eventually, we decided the two activities were similar enough that a single format would suffice. Nevertheless, linking and execution focus on different aspects of the program, and ELF provides parallel views of the file contents.



An *ELF header* resides at the beginning and holds a "road map" describing the file's organization. This control information includes the locations and sizes for the *section header table* and the *program header table*. In turn, these tables contain information describing the logical units for each view: sections for linking and segments for execution. Each table is optional for the alternative view. Although the figure shows the program header table immediately after the ELF header, and the section header table following the sections, files may differ. Sections and segments have no specified order; only the ELF header has a fixed position in the file. Further explanations appear below.

Data Representation

ELF uses six basic objects to define its data structures: unsigned bytes and five other integral types. The sizes and alignments appear in the table below for the 32-bit class.

All data structures that the object file format defines follow the "natural" size and alignment guidelines for the relevant class. If necessary, data structures contain explicit padding to ensure 4-byte alignment for 4-byte objects, to force structure sizes to a multiple of four, etc. Data also have suitable alignment from the beginning of the file. Thus, for example, a structure containing an `Elf32_Addr` member will be aligned on a 4-byte boundary within the file. Other classes would have appropriately scaled definitions. To illustrate, the 64-bit class would define `Elf64_Addr` as an 8-byte object, aligned on an 8-byte boundary.

Following the strictest alignment for each object allows the format to work on any machine in a class. That is, all ELF structures on all 32-bit machines have congruent templates. For portability, ELF uses neither bit-fields nor floating-point values, because their representations vary, even among processors with the same byte order. Of course the programs in an ELF file may use these types, but the format itself does not.

File Identification

As mentioned above, ELF provides an object file framework to support multiple processors, multiple data encodings, and multiple classes of machines. Although the initial implementation of ELF supports 32-bit architectures, it is intended to be extensible to larger (or smaller) architectures. To support this object file family, the initial sixteen bytes of the file specify how to interpret the file—independent of the processor on which the inquiry is made, independent of the processor on which the file was created, and independent of the file's remaining contents.

A file's first four bytes identify the file as an ELF object file; their respective values are `0x7f`, `'E'`, `'L'`, and `'F'`. Another byte defines the file's class, or capacity. Keeping this in the machine-independent header lets the file format support various architecture families, without imposing the sizes of the largest machine on the smallest. The 32-bit class supports machines with files and virtual

Name	Size	Alignment	Purpose
`Elf32_Addr`	4	4	Program address
`Elf32_Half`	2	2	Unsigned medium integer
`Elf32_Off`	4	4	File offset
`Elf32_Sword`	4	4	Signed large integer
`Elf32_Word`	4	4	Unsigned large integer
`unsigned char`	1	1	Unsigned small integer

address spaces up to four gigabytes.

A byte order descriptor also appears in the initial sixteen bytes. Currently, we allow two's complement big-endian and little-endian encodings, and adding new encodings is simple. The data encoding controls the interpretation of the remainder of the file. That is, the rest of the file uses the natural data encoding for the target processor, regardless of the machine on which the file was created. Recording the file's byte order explicitly lets one unambiguously—and portably—interpret ELF file contents.

Finally, the initial header holds the file's version number. We needed a way to extend various control structures as the need arose. Although we could have allocated padding space in the structures, we did not know how much padding to use. Having too little defeats the extensibility goal; too much padding makes files bigger than necessary. The version number implicitly defines control structure sizes. If a structure must grow, we shall add the information and change the version number.

ELF Header

Besides the sixteen identification bytes described above, an ELF header contains control information for the entire file. Knowing some processors would have special needs, we defined ways to accommodate those requirements. The System V ABI defines object file characteristics that all implementations must follow; the ABI processor supplements give the machine-specific details. I'll mention the more important examples of this as we go along. To assist the linking and execution views, we placed common information in the ELF header, including the following.

- *Object file type.* Current types include relocatable file, executable file, and dynamic library. Additionally, the "core file" (created to hold the image of an abnormally terminated process) has a defined type. Using the same file format for germinal and deceased program images has advantages for debuggers and similar programs.
- *Machine.* This value specifies the required architecture for an individual file. Each target processor has a distinct value.
- *Entry point.* This member holds the virtual address to which the system first transfers control, thus starting the process.
- *Section information.* Under the linking view, an object file is a collection of sections, each described by an entry in the section header table. The ELF header gives information describing the table itself, including the location in the file, the number of entries, and the size of each entry. The table is contiguous; so the product of the entry size and entry count gives the total table size.
- *Segment information.* Similarly, the execution view provides a collection of segments, visible through the program header table. The ELF header describes the program header table by giving its location in the file, the number of entries, and the size of each entry. This table also is contiguous.
- *ELF header size.* Among other items, the ELF header includes its own size in bytes.
- *Flags.* A file has a word of flag information in the initial header. Each processor can have its own semantics for the values.

Information in the ELF header is defined in terms of the basic objects discussed above. Thus for the 32-bit class, the structure uses `Elf32_`*type* values. Other structures, such as the section and program headers, also follow this convention.

Sections

With three exceptions (the ELF header, the program header table, and the section header table), sections hold all information in an object file. To enforce consistency in the linking view, ELF defines several section constraints. First, every section has exactly one entry in the section header table. Section headers may exist that do not have a section. Second, each section occupies one contiguous (possibly empty) sequence of bytes. Third, sections may not overlap; so no byte resides in more than one section. Finally, an object file may have inactive space. That is, the various headers and sections might not cover every byte in the file. The contents of the inactive space are unspecified.

Notice that ELF does not designate fixed areas of a file for such standard facilities as the symbol table, relocation records, and so on. Instead, the file format allows one to define section types, with associated semantics. I'll discuss this more below. A section header characterizes its associated section as follows.

- *Section index.* A section header's table index provides the internal connection mechanism between sections. That is, if one section must refer to another, it uses the section index as the key. For the 32-bit class, we reserved 257 section numbers for special purposes, allowing up to 65,279 sections. We could have used a 4-byte index, but a 2-byte index significantly reduces file size. Typical files have fewer than 20 sections, so the current limit seems safe.
- *Name.* Sections may have arbitrary length names, contained in a "string table" section. The ELF header holds the string table's section index, selecting one of possibly many string tables for section names.
- *Location.* Each header also tells where to find the section in the file, by specifying the

section's file offset and size.

- *Address and alignment.* If a section contributes to the process image during execution, its section header will contain the virtual memory address at which the section resides. Moreover, some sections hold data that require special alignment for hardware or performance reasons. For example, many machines require double-precision floating-point values to be aligned on 8-byte boundaries. The section header records the alignment to let programs preserve the constraints.
- *Type and entry size.* A section's type specifies the structure and semantics of the data it contains. If the section holds a table, the section header tells the size of each entry. Section types are divided into three ranges: one for types common to all implementations, another for processor-specific sections, and a third available to applications. More information on section types appears below.
- *Flags.* Each section has a word of flags, with some bits reserved for processor-specific semantics. Currently, the three defined flags affect a section's execution behavior: one bit requires write permission, one requires execution permission, and one specifies whether the section must reside in a loadable segment during execution.
- *Connection information.* Finally, a section header has two overloaded values whose meanings depend on the section type. The values can record connections or other special information. For example, a relocation section's header holds two indexes: one for the section to be relocated, another for the symbol table that the relocation entries reference. In all honesty, connection handling is one of the least satisfying parts of ELF, because of the type-specific meanings. Still, it does what we needed, and we could not devise a better solution.

One can arrange or rearrange ELF sections in a file, and the format lets one define semantics through section types. Currently, we have about a dozen section types, including symbol table, hash table, string table, program instructions and data, and relocation. Some, such as the symbol table, have system-defined formats and meanings; others have structures and meanings assigned strictly by the application; still others have machine-specific meanings defined in a processor supplement to the ABI.

Sections have meanings because programs that read and write object files collectively agree on types' semantics. If an existing section type changes, programs that deal with that section will have to adapt to the new definition; this can happen two ways. One can extend existing sections, using the version mechanism and the section header's entry size. Alternatively, one can define a new section type to replace a type that is no longer adequate. Over time, object file producers will simply stop creating the old type, and object file consumers will learn to understand the new. Obviously, this will be inconvenient if a common section (such as the symbol table) changes. Nonetheless, having a facility for change is as important as using it judiciously.

Segments

Because linking and program execution have different needs, sections and segments follow different constraints. As execution entities, the program header table and segments give information to let the system run a program. Unlike sections, segments may contain any information in the file, including control headers, multiple sections, portions of sections, and parts of other segments. Although a segment must be contiguous within the file, it has few other constraints. Entries in the program header table describe segments.

- *Type.* About half a dozen segment types give information to the system for program execution. A ''loadable segment'' is the most common (and most obvious); other types also assist program launching. A range of types is reserved for processor-specific use.
- *Location and size.* As one would expect, the program header gives the segment's offset and size in the file. An additional memory size says how much space to allocate for the execution image, thus giving a simple way to represent uninitialized data without consuming file space. (If the memory size exceeds the file size, the system allocates additional memory for the segment and sets it to zero.)
- *Address and alignment.* Segments may have a virtual address, giving the required position in memory. The segment alignment preserves its sections' alignments, as discussed above. If necessary, a segment may have stricter alignment for paging or other run-time issues.
- *Flags.* Finally, a segment has a word of flags. Currently, loadable segments have bits to control the required memory management permissions (read, write, and execute). Four flags are reserved for processor-specific meanings.

When building an executable file, the link editor groups sections into segments, with the exact rules tailored for individual processors and memory management units. Typically, the link editor builds two loadable segments: one with write permission and one without. Individual sections (such as text, read-only data, and writable data) retain their section identities, and the link editor places them in segments that have the necessary attributes for program execution.

ACCESS LIBRARY

Because the object file format specifies the layout of a file, it defines data structures, alignment, byte order, and other file properties. These definitions are appropriate for the target processor upon which the program is intended to run. Compilation system programs might not match the object file constraints, especially when running in a cross environment. ELF's design explicitly acknowledges the distinction between the external (file system) and internal (memory) representations of an object file. In parallel with defining the object file format, we also defined an access library that provides this internal view of the external file contents. Although the library provides some incidental services, its main purpose is to translate object file information between the internal and the external representations.

As discussed previously, ELF builds its structures with several basic types. Knowing the external representation, the access library converts to or from the internal representation as necessary. For example, the external representation of `Elf32_Off` is a 4-byte unsigned word, in a supported data representation. Internally, `Elf32_Off` can be any type that supports the required semantics. Internal and external representations might be different, but the access library hides this from a running program.

While writing the access library for the 32-bit class of files, I ported the code to a 64-bit machine (a CRAY X-MP). The native C compiler used 8-byte objects for `short`, `int`, and `long`; so the natural implementation of ELF's 2-byte and 4-byte types used 8-byte objects internally. I wrote some tests to create both big-endian and little-endian files with known values for all the ELF basic types and structures. Without changing a line of the library or test code, I compiled and ran the tests on the CRAY and various 32-bit machines. The external files on all machines matched exactly, even though the internal CRAY data matched neither the sizes nor the data representations of the external files.

Following standard practice, a header file supplies common ELF definitions, including the current version number. If the structures must change, the header file's version number will change, the library will support both the new and the old versions, and new programs will see the new information. A program that uses the access library must specify what file version it wishes to use. In effect, it tells the library what header files were used to build the program. When exchanging structures with the application, the library uses data that match the program's version. Version translation resembles the internal and external conversion described above, and we expect it to help maintenance in the future.

EVALUATION

This section compares ELF with previous formats and shows how various decisions contribute to the goals discussed above. I have already described some issues, such as supporting multiple processors.

Data Representation

Previous object files typically began with a 2- or 4-byte "magic number," intended to distinguish them from other text or data files. Magic values were chosen not to be printable ASCII, and they were encoded in the standard byte order for control information. Depending on the format, the magic number identified either the target processor or the file type. Several problems arose with these policies.

First, the classic *a.out* format did not record the target processor. Thus different machines could—and often did—use the same magic number to mark executable files. If two different machines had the same byte order, one operating system might execute a program intended for the other processor. It usually didn't take long to hit an illegal instruction, but the problem could be mystifying at first. Identifying the proper target for an executable file could be equally puzzling. Some vendors extended the classic *a.out* format to include the processor, but there was no standard assignment of values to processor types.

If a program examined a file on a machine with the wrong byte order, the magic number might look bad. Similarly, if an old program examined a file with a new magic number, the magic number might look bad, even though the program could have understood the rest of the file. Some programs refused to work on unrecognized files. Others tried various byte swapping spells, hoping for a known effect on the magic bytes.[4]

COFF used a hybrid byte order. Sections of the file (such as text and data) used target byte order, while control information (such as the headers, symbol table, and relocation) used the byte order of the machine used to create the file. When transferring a COFF object file from one machine to another, a special program was needed to convert the control information to the format of the receiving machine. That is, COFF maintained a consistent internal representation by changing the external representation. Ostensibly, this design made cross development simple, but it defeated file sharing among heterogeneous machines.

Previous byte swapping heuristics subtly constrained magic number selection. I recall one amusing incident about ten years ago when someone innocently used (octal) `0401` as a magic number. At the time, we were using little-endian development machines, big-endian target machines, arcane file

[4] I think of this as the sorcerer's apprentice approach.

conversion procedures, and a slow network. Because the magic number was the same on big-endian and little-endian machines (0401 equals 0x0101), the value always looked right, whether the file had been properly converted or not. After considerable confusion, the number was reassigned to an asymmetric value, and the new value's reflection was reserved as well.

ELF's machine-independent header unambiguously lets a program identify a file as one it can or cannot read. The explicit class and byte order descriptors tell what data representation the file uses, and the file's external representation stays constant regardless of host and target combinations. Because the access library translates between the external and internal representations, one can write portable programs that can ignore byte order differences for cross environments. Additionally, one can write programs that work in both native and cross environments. Many of our object file programs work on any ELF file, regardless of the target processor. This can simplify cross environment support, because it reduces the number of architecture-specific development programs. In other words, ELF maintains a constant external representation to allow file sharing, and the access library provides a consistent internal representation to support native and cross development.

Although ELF uses target format for the bulk of the file, we considered using a machine-independent representation for some control structures. This also would have met the file sharing and portability goals. Moreover, it would have simplified the access library somewhat, by having a single external representation rather than one per target byte order. On balance, though, target byte order seems to have been the right choice.

Assuming native development is the typical case, one would like to make it as efficient as possible. Target byte order makes the internal and external representations equivalent; the access library recognizes this and avoids unnecessary conversion. Using a machine-independent byte order either would have unnecessarily burdened all development, or it would have favored machines that happened to match the canonical byte order. ELF supports cross development as a common activity, but all native development enjoys a natural performance advantage.

Target format also makes program execution more efficient, especially during dynamic linking. To illustrate, we wanted the link editor and the dynamic linker to share the same symbol table to avoid duplication. If the symbol table used a machine-independent byte order, dynamic linking would consume more execution time. The dynamic linker could convert the symbol table once, but that would cause many page faults and further delay initialization. On the other hand, if it decoded each referenced value as needed, the code would have been clumsy and error prone (forgetting a single conversion could lead to disaster). Target byte order simplifies the code and lets the dynamic linker spend its time building the process image, rather than shuffling bytes.

Version Handling

Given a known version number, the explicit structure sizes mentioned above are superfluous. One is tempted to conclude that either the file should use a version number, or it should use explicit sizes. ELF uses both for the following reasons. We anticipate the need to change versions someday. It might not happen soon, and we'll doubtless put it off as long as possible. But someday, for some reason, we shall have to grow an existing control structure and increase the version number. Our intent is to extend ELF in upward compatible ways. The new version will maintain previously defined information, thus preserving existing semantics.

After the version changes, both new and old object files will exist. Perhaps more important, old programs will exist that understand only the old version. If the structure sizes were tied only implicitly to the version number, an old program might be disabled. Not recognizing the new version number, it would not know the implicit sizes and thus would be unable to examine tables with multiple entries. With explicit sizes in the file, an old program should be able to examine a new file and ignore the extra information. Of course, it would not provide the new functionality, but it could reliably examine the information for its own version.

A similar argument applies to new programs looking at old files. The treatment of missing information depends on context and will be specified when and if extensions are defined. Conceivably, a version number might change without associated structure growth. Without an explicit version number, programs might be unable to detect the semantic change. Though somewhat different in flavor, the version number, explicit sizes, and type values resemble self-describing data [4].

Version handling further highlights the close association between the file format and the access library. From the application's view, the internal format always uses the application's version, regardless of the external file's version. Following our general philosophy, we took additional complexity into the single library to simplify multiple applications.

Adaptability and Flexibility

Previously, program loading information could reside only at the beginning of the file and could accommodate only two loadable segments. Although fixing the position makes the data easy to find, the primary performance benefit occurs because the data are in the initial disk block. ELF's program header table has neither a fixed position nor a fixed size and thus eliminates the restrictions. Nonetheless, the table typically follows the ELF header immediately, providing the performance advantage of being in the first block. In other words, we have policies to favor common activities, but if other activities become more important or if we discover better policies, ELF provides the flexibility to change.

Loadable segments also suffered restrictions on their contents, because they could hold exactly one linkable section. To see how this was a problem, consider dynamic linking, which needs symbol and relocation information at run-time. Because previous object files' symbol tables and relocation could not reside in sections, the dynamic linking information had to be duplicated and merged into an existing section to appear in the running program image. Clearly, duplication makes files bigger, and merging obscures the identity of the extra data. ELF uses sections to hold symbol tables and other information that used to be hard-wired in the object file format. Moreover, an ELF segment can comprise any contiguous sequence of bytes in the file. For dynamic linking, we simply placed the symbol table and relocation sections in loadable segments. From a linking perspective, the sections retain their identities and semantics, and they can simultaneously contribute to the executable program image.

Various control structures, such as symbol tables, relocation, and section headers also had fixed contents, making it difficult to improve performance or functionality. For example, COFF defined a 10-byte relocation structure with several problems. First, it was larger than necessary for typical CISC processors. Second, it was inadequate for RISC processors. Third, the 10-byte external structure occupied 12 bytes on most internal representations, making relocation clumsy and inefficient to process.[5] Fourth, the file format allowed neither growth nor redefinition; so one could not fix the first three problems. ELF currently defines two relocation structures: one with 12 bytes for typical RISC machines, and one with 8 bytes for other architectures. People porting ELF to a new machine simply choose the relocation format that works better. If convenient, they can use multiple formats; if necessary, they can define a new one. Similarly, ELF allows extension and redefinition for other control structures.

COFF symbol tables had C-specific debugging entries, with no provision to extend or replace them for other languages. Even the C support was limited, being unable to represent ANSI C concepts such as `const`, `volatile`, and function prototypes. In parallel with specifying the file format, we designed a section to hold debugging information. Although it initially supports C and C++, it provides a multilingual framework. As with other sections, ELF will support changes if compiler and debugger writers devise something better. Finally, the debugging section can define its own symbols and use relocation to reference symbols' values, giving data structuring capabilities.

Access Library

Initially, some people were apprehensive that the access library could not give good enough performance. COFF had a library, for example, and programs such as the COFF link editor manipulated files directly because the library was too slow. Nonetheless, we knew that an access library was the only practicable way to accomplish the goals we had set. All the ELF programs used the access library from the outset, despite the unanswered performance questions.

After writing the library and tuning it on synthetic tests, I analyzed the COFF and ELF link editors' performance. First, I compiled test applications to get two sets of object files, one ELF and the other COFF. Second, I linked the applications and measured the user and system time consumed.

As one can see in the table, the ELF linker ran significantly faster than the COFF linker, each processing its native input. I attribute improved performance to several factors.

First, the interfaces and implementation assume a large virtual address space. This lets the library and applications deal with large units (files and sections), reducing overhead and taking advantage of the system's virtual memory. Second, the library uses memory-mapped files if available and otherwise simulates them with direct I/O calls. This proved more efficient than reading entire files or using buffering I/O packages. Third, ELF files tend to be smaller than the COFF equivalents, although this is not always true (ELF files smaller than a few

Experiment	File	Input Size	Output Size	Seconds	Normalized Time
63 input files	COFF	199,221	176,017	4.86	1.00
	ELF	200,876	157,188	2.75	.57
99 input files	COFF	570,754	524,464	12.79	1.00
	ELF	518,624	433,240	7.09	.55

[5] The COFF symbol table entry was 18 bytes long and had similar problems.

thousand bytes are sometimes slightly bigger than COFF). Especially for large files, ELF's size advantage helps performance. Finally, ELF's internal and external representations match for native development, eliminating problems such as the 10-byte COFF relocation entries mentioned above.

Translation is convenient and necessary for cross environments, but we wanted to eliminate execution overhead for native development (when the internal and external formats match). Thus the library determines at run-time the data representation of the current processor and avoids superfluous conversions. During our development, this caused an interesting "problem." We were using a cross environment on a mainframe. After several weeks of development (including building programs and running regression tests), some people were tuning the link editor performance. They noticed time going to internal ELF library routines and asked me about it. We found that a mainframe compiler error had disguised the native data representation, causing the library not to recognize the equivalence between the mainframe and the target data representations. Except for using more processor time (which had gone unnoticed for weeks), the extra translation had been transparent to the development environment.

COMPATIBILITY ISSUES

This section describes how we dealt with the practical problems of introducing a new object file format. First, people obviously need to execute their old programs. This can happen by letting the kernel continue to understand the old format. People also want to preserve relocatable object files, including libraries and device drivers. We considered this essential, because some people might not be able to regenerate their object files. For example, someone who purchased database libraries, device drivers, or other relocatable object code might not have access to the original source code.

We built backward compatibility into the access library. Whenever a program opens an object file with a previous format, the library presents the file to the program as ELF. In other words, the access library translates the foreign external format to an ELF internal format (without changing the original file), and new programs deal exclusively with ELF. Other object files make programs manipulate the external format directly, implicitly assuming equivalence with the internal format. I've discussed the problems this causes for cross development, and backward compatibility illustrates another shortcoming. Without the access library's conversion, programs would have had to deal with the new and old formats, duplicating code and complicating maintenance. Although the new programs support files in the old formats, they can be written as if the previous formats had never existed.

After implementing the COFF translation, I repeated the performance tests described above for the access library, except I gave COFF input to the ELF linker.

Even with the access library converting the COFF files to ELF internally, the ELF linker processed COFF input about 20% faster than the native COFF linker. Interestingly, the experiment approximates ELF cross environment performance, assuming ELF-to-ELF translation is no more expensive than the COFF-to-ELF measured here.

Conversion overhead was low, but we also wrote a program to translate old files to ELF permanently. This program, called `cof2elf`,[6] simply opens a COFF input file, lets the library convert the internal representation to ELF, and then tells the library to write the result back to the file system. Except for error checking, the program is almost trivial. By using this permanent conversion facility, people can avoid COFF translation overhead for production work.

SUMMARY

ELF provides several significant improvements to the UNIX system's object file facilities.

■ The access library clearly separates the internal and external representations of the file. Previously, object file programs had embedded knowledge about byte order, host and target architectures, and external file format. These programs were not portable and frequently had subtle errors. The combination of the ELF format and the library's translation facility makes application programs portable, regardless of host and target machine combinations.

■ ELF provides a portable framework to support multiple processors and multiple classes

		Input	Output		Normalized
Experiment	Linker	Size	Size	Seconds	Time
63 COFF input files	COFF	199,221	176,017	4.86	1.00
	ELF		160,508	3.70	.76
99 COFF input files	COFF	570,754	524,464	12.79	1.00
	ELF		480,016	10.30	.81

[6] A diminutive, mischievous, magic-wielding person of Germanic ancestry stole an *f* from `cof2elf`.

of machines. Five basic object definitions identify a file class: memory address, file offset, small unsigned integer, large unsigned integer, and large signed integer. A 32-bit class exists, and other classes can be created. A file's machine-independent header lets a program identify and read the rest of the object. Processors can define their own information, such as relocation types, which are inherently machine-specific.

- Both the external format and the access library support extensions. New information can be defined or added to existing structures. Moreover, existing (old) programs can continue to operate in the face of unknown (new) information. Even important structures, such as symbol tables, can change under the ELF framework.
- An ELF file offers flexible control over internal organization. Information such as symbol tables, program code, or operating system information can be arranged as necessary for improved efficiency and convenience. Our regular policies favor program execution performance, but the object file itself can support other policies.
- Finally, an individual ELF file always has the same external representation. This binary portability is prerequisite to convenient file sharing in heterogeneous networks. In combination with the access library, ELF gives consistent internal and external representations, significantly improving the portability and transportability of programs.

Taken individually, the deficiencies noted above were merely inconvenient, and we couldn't justify the expense of introducing a new format without clear benefits. Moreover, the problems surfaced gradually, as the computing environment changed. Byte order issues, for example, generally are benign in strictly native environments; cross development and heterogeneous networks make the issues unavoidable. The prior formats' cumulative limitations and irritations, including their resistance to change and evolution, finally convinced us to introduce a new object file. We believe ELF will mitigate this mischievous misoneism.

ACKNOWLEDGMENTS

Many people worked on ELF's design and implementation. The following people made especially noteworthy contributions: Elizabeth Flanagan, Rob Gingell, Barbara Howe, David F. Prosser, Joel Silverstein, and David L. Weaver.

REFERENCES

1. AT&T. *System V Application Binary Interface.* Englewood Cliffs, NJ: Prentice-Hall, 1990.
2. S. C. Johnson and D. M. Ritchie. ''Portability of C Programs and the UNIX System.'' *The Bell System Technical Journal* **57**, No. 6 (July–August 1978), pp. 2021–2048.
3. AT&T. *System V Interface Definition, Third Edition.* Copyright 1989.
4. Jon Bentley. *More Programming Pearls.* Reading, MA: Addison-Wesley, 1988.

James Q. Arnold is a Distinguished Member of Technical Staff in the Application Integration Department of AT&T Bell Laboratories. His interests include compilation systems, development environments, and program portability.

USENIX Summer Conference
June 11-15, 1990
Anaheim, California

Issues in Shared Libraries Design

Marc Sabatella – Hewlett-Packard Company

ABSTRACT

On traditional UNIX systems, executable programs may be shared on disk and in memory, but library code may not be shared between programs. Several UNIX platforms now support shared library facilities. While the existing implementations share some common aspects, there are several important concerns that have been addressed differently. These include *program organization*, *sharability*, *library binding time*, *importability*, *symbol binding time*, *binding granularity*, and *version control*. I explore these issues and their impacts on program size, execution speed, flexibility, transparency, reliability, and security. The problems are interrelated, and often there are tradeoffs against linker and loader complexity as well. I then give an overview of the HP-UX shared libraries design, emphasizing how we resolved the concerns discussed in this paper.

Introduction

The traditional UNIX philosophy encourages the use of small utilities that may be combined in various ways. Any given program such as *grep* may be used in several different applications; this constitutes one form of sharing. Most UNIX systems allow processes to share memory images of programs as well as disk images; this constitutes another form of sharing. However, small utilities like *cat*, *wc*, and *diff* consist mostly of library code that is duplicated in each program. X window programs typically contains large amounts of X library code, much of which is duplicated in every X program. Thus there is often a substantial amount of code that is duplicated several times. This duplication will exist both in the disk images of the programs and in the memory images of the processes executing the programs.

Therefore it is desirable to be able to share library code between programs. Sharing library code can decrease disk space and memory requirements. There have been several attempts at implementing shared libraries on UNIX systems, as well as other systems. While there are aspects common to all the designs, there are several key issues that must be addressed by any implementation. These concerns include *program organization and sharability*, *library binding and loading time*, *importability*, *symbol binding time*, *binding granularity*, and *version control*. The resolutions chosen can affect program size, execution speed, flexibility for the programmer, flexibility for the application user, transparency, reliability, and security.

In this paper, I examine the impacts of each of these issues, explore the various alternatives and tradeoffs, and occasionally make value judgements. I then give an overview of the HP-UX shared libraries design, emphasizing how it addresses the concerns raised in this paper. The version control and binding granularity schemes are discussed in more detail, since these features are unique to our design.

Issues

The issues discussed in this paper are often interrelated. They are presented in order from the most fundamental to the more specific. The ordering does not necessarily imply anything about their relative importance.

In the the following discussions, the term *static* refers to operations that occur when building a program, whereas *dynamic* refers to operations that occur when executing it. *Linking* refers to at least three separate concepts: *combining* object files, *resolving* symbolic references, and *relocating* code so that it may run at a particular address. Linking is performed by the UNIX utility *ld*. *Loading* is the act of bringing a program into the address space of a process so that the program may be executed. Loading is performed by the system call **exec()**.

Program Organization And Sharability

In traditional UNIX, a program consists of one or more source files that may call functions and refer to data defined in system libraries, such as the C library, the math library, and the X library. The source files are compiled separately and linked together along with any libraries they reference. The linker includes the referenced code and data from each library, as well as the actual program code, in the final executable file. Thus, a copy of a library routine exists in the library and in every executable that uses the routine.

Sharing an object between two programs on disk means that only one copy of the object exists on the disk — the object is not copied into every program that uses it. There are two basic requirements for sharing an object between two process in memory. First, it must be possible to recognize that the two processes are indeed referring to the same object. If the object is shared on disk by storing it in one central location with a fixed filename, and the two programs both refer to that same filename, then this recognition is easy. Second, the object must appear the same to each process that uses it. For a library routine, this implies that the text segment must be pure, in the sense that it is not self-modifying, and that any absolute address references it makes are valid for each process. This can be achieved for internal references by either fixing the absolute address at which the segment resides so that it is the same in each process, or by using *position independent code* (PIC). Pure text can be achieved for external references by either fixing the addresses of the external locations or by using some form of indirect addressing.

There are problems associated with using absolute fixed address for shared objects. If the addresses are fixed by the person building the object, there is the possibility that two objects will be built with overlapping address requirements. For SVR3, AT&T tried to partition the address space and establish guidelines for the allocation of partitions between libraries [2], but this type of approach is an administrative headache. Another alternative is to allow the system administrator to fix the addresses of shared objects present on his system.

In general, position independent code and indirect addressing are slower than ordinary code on most machines, and it usually requires special effort or compiler modifications to generate.[1] Position independent code has the advantage of being executable regardless of the address at which it is loaded, thus eliminating the problem of assigning addresses to libraries.

Sharing in traditional UNIX systems is on the program level. Two processes using the same program may share the complete disk and memory images of the text segment, because the system can easily recognize that the two processes are using the same executable file, and the text segment is completely contained within the file. The address of the segment is fixed (usually at zero) and it makes absolute external references only to its own data and bss segments, which are at fixed addresses (usually immediately following the text segment).

To implement shared libraries, it is necessary to achieve a finer granularity of sharing. One extreme example of this is in the Multics operating system [6], in which there is no static link phase or distinction between program and library. Each individual object file is individually sharable. At the other end of the spectrum is a design implemented for the Softbench product on HP-UX [12], in which all the libraries on the system are linked into one executable file that can be shared by programs.

The approach taken in shared libraries implementations on systems such as VMS [10], MPE [4], Domain [5], and OS/2 [11], as well as on UNIX systems such as SVR3 [3], SunOS [9], and SVR4 [1], is to share at the library level. There is a file corresponding to each library on the system, and programs can share each library individually. VMS and SVR3 assign fixed addresses to achieve pure text, while the other systems use some form of PIC or indirection. SunOS does not require PIC, but if it is not used, the resultant library will not be pure and cannot be shared in memory. In each case, a library is created by linking together the object files that compose it. I will be concentrating on this model, since it most closely resembles the traditional UNIX model. Tradeoffs between the approaches will be discussed later.

In each of the implementations mentioned above, the loader maps the program and all required shared objects into the address space of the process. The text segments of objects may be shared, but the data segments normally are not. Private (per process) data is usually implemented through *copy on write* mapping in the virtual memory system. The data segment may contain variables and tables initialized with absolute addresses, in which case dynamic relocation may be required.

Library Binding And Loading Time

In traditional Unix systems, routines in libraries can refer to data and other routines in the same library or in other libraries. Certain conventions and standards are usually followed. For example, if a library provides a definition of a routine called **malloc()**, the library developer should make sure the definition conforms to the specifications of the standard version. The developers of the X library may include calls to **malloc()** without knowing if the call will resolve to the standard definition in the C library, or to a functionally equivalent version such as in a profiling version of the C library.

The application programmer has complete control over how his function calls are resolved. By linking with the X10 library instead of the X11 library, he can make sure that his programs will run on X10 systems. By using the profiling version of the C library, he can collect data on the run time behavior of his program. In any case, the code from the libraries he selects is statically linked into his program, so if he wishes to change libraries, he must relink the application.

[1] On segmented architectures, the distinction between absolute and position independent code is not always clear. There are almost never absolute addresses in compiler generated code for such architectures; addresses are usually represented as offsets from a base register. A text segment can be executed unmodified at any absolute address by simply setting the base registers appropriately.

This model may be applied to shared libraries as well. When building a shared library, the library developer would make standard assumptions about the resolution of external references. The application developer would list the libraries to be used on the linker command line, and the linker would arrange for the loader to load those exact libraries at run time. There would be no differences between program development with shared libraries and program development with archive libraries.

Run Time Library Search Paths

Recording the fully qualified pathname for each library at static link time could cause problems for system administrators. For example, one common method of implementing mixed-architecture clusters is to create separate **/lib** and **/bin** directories for each architecture. Each user's search paths must be set appropriately for the machine on which he is running. Thus the C library might exist in **/lib/libc.a** on a standalone machine of type "a", but in **/lib/machine-a/libc.a** on the same type of machine in a mixed cluster. If the full pathname of the C library were recorded in the program, then the program could only run on similarly configured systems.

A simple workaround is to allow the loader to search a library path at run time. Thus only the name **libc.a** would be recorded in the program, and each user's run time shared library search path would be set to the appropriate search path for the machine on which he is running.

There is a potential security problem with this arrangement. Users must not be allowed to override the run time library search path for libraries associated with trusted programs. One workaround is to require all trusted programs to be built with archive libraries. Another is to hard code library pathnames in trusted executables. The most general solution is to partition the run time search path into a system wide search path and a user path. The user path may be implemented as an environment variable; the system path should be modifiable only by the system administrator. Trusted programs should be flagged so that the loader will ignore the user library search path.

Run Time Library Installation

Run time search paths take away some of the application developer's control over which libraries are used by his application. The VMS, MPE, and Domain operating systems allow system administrators or users complete control over the locations, names, and number of shared libraries used by a program. Some libraries are globally *installed* (made available to all users) at system boot time, and users can also locally install libraries for their own use. The application developer can also specify libraries at link time to be installed when the program is run. In Domain, once a library is installed in a process, it is available to all descendent processes as well. Thus installing a library into a shell process makes it available to all applications invoked from that shell. Ideally, for security purposes, trusted programs should be linked in such a way that only globally installed libraries are used at run time.

Symbol Namespace

A static link phase should be available to allow the application developer to specify special versions of library entry points. If a program depends on a library that is unlikely to have been installed in a user's process, the developer must be able to cause that library to be loaded. For example, a program may depend on characteristics of math routines defined in a special math library that takes advantage of optional floating point hardware. If someone tries to use the program without installing that library, strange things may happen as the program is forced to use the standard math library version of those routines, unless name of the special math library is recorded in the program.

There are other problems related to symbol namespace. For instance, there may be several libraries that all define an entry point called **init()**. The application developer must be able to specify which library he intended to contribute that definition, since the various versions are almost certainly not interchangeable. Note that dynamic type checking is not enough — many of the definitions may have the exact same type signature. Explicitly specifying libraries during the static link phase is one way to reduce the scope of the namespace and thereby control which definition is chosen.

Another method of reducing the global namespace is to associate *package* names with each entry point. Functions relating to C standard I/O would go in one package, trigonometric functions in another, and so on. A library could contain several packages, and there could be definitions for a given package in several different libraries, as long as all packages with the same name were compatible. This reduces the global namespace problem to the two smaller problems of assigning package names and assigning symbol names within a package. Both the function name and package name would be recorded in the program for each library entry point referenced. It should thus be an easier matter to make sure the correct definition is used at run time. This approach is being investigated by OSF [8].

Library Load Time

In any of the organizational models, regardless of when the library name binding is established, the question of when to *load* the library is relatively unimportant. Loading a library consists of mapping it into the address space, which is a cheap operation on most UNIX systems. Loading all required shared objects at process startup time is usually acceptable. The alternative is to wait until an object is referenced before loading it. This leads to the related, and much more important, question of how and when to bind the external references into and out of a library. This issue will be discussed in the next few sections.

Importability

Using fixed addresses or position independent code for shared libraries ensures that references to internal items will be pure. When referring to external items, these mechanisms must be augmented.

Fixed External Addresses

If fixed addresses for libraries are used on the system, the simplest solution is to assume, when fixing the address of a library, that any other libraries referenced can also be permanently fixed now, and that the relative offsets of all their entry points are fixed as well. Thus the X library could refer to **malloc()** in the C library simply by using its absolute address. SVR3 and VMS used this approach, and suggested the use of jump tables at the start of each library so that relative offsets of the actual routines could change as long as the jump tables were kept consistent. Maintaining relative offsets of data items is more difficult. If the addresses are fixed by the system administrator instead of the library developer, then jump tables are not needed. If addresses within any library on the system change, the system administrator can relocate the other libraries that depend on it. Any application programs that depend on those absolute addresses will also have to be relinked, however. Furthermore, applications may not be portable across machines, since libraries may have been fixed at different addresses by different system administrators.

In addition to the administrative problems, there are other flaws in this approach. First, it works only if system wide fixed addresses for libraries are used. If this is not the case, then another mechanism is needed. It was observed earlier that normally, a library can be written without full knowledge of how its external references will be resolved. For example, the math library can call **matherr()** without knowing if the definition will be supplied by the standard C library, by a profiling version of the C library, or by a user program. It is the application developer (or possibly the application user) who makes this determination. If the call to **matherr()** is fixed to an absolute address, then overriding the definition for individual programs or users is not possible without destroying the pure text property of the library.

Linkage Tables

Because of the problems associated with statically resolving imported references to absolute addresses, it is desirable to statically resolve such references to a *linkage table* and allow a dynamic linker to bind the symbolic references for the table at run time. The dynamic linker in Multics was incorporated into the virtual memory system, and required support from the underlying architecture. On UNIX systems, dynamic linking functionality may be built into the loader (as in Domain) or invoked as a separate function (as in SunOS).

Each linkage table entry corresponds to an imported symbol. The table must be private per process, since different processes using the same library may provide different definitions for particular imported symbols. There is usually one linkage table per library, so that each library can easily refer to its own linkage table entries. It is usually included in the library data segment, so that the dynamic linker is free to modify its contents on a per process basis. The symbol table associated with linkage table may be combined with the standard name list for the object, but since it must be available for the dynamic linker at run time, it is often included as a separate *import table* within the text segment.

The linkage table may consist of two separate sections, one for procedure references and one for data references, but it is conceptually one table. Procedure calls may be resolved directly to the table entry for the corresponding symbol. Thus the procedure section of the linkage table would be implemented as a jump table. Data references must be explicitly coded (or compiled) to access data indirectly through the linkage table entry for the corresponding symbol. Thus the data section of the linkage table may be simply a table of absolute addresses.

In general, neither the library developer nor the compiler will be able to determine if a particular reference in a library will be imported or not. For example, a function within the C library may call **malloc()**, but it is not known when building the library whether the reference will be resolved to the definition within the C library, or whether a particular application program will supply its own definition. It is important to be consistent in this regard. If the C library assumes its own definition will be used, but the application program uses another definition, the heap may become corrupted at run time, since many memory allocation schemes are mutually incompatible.

Instead of forcing the library developer to identify possible imports and treat them specially, it may be advisable to treat all references as potentially imported. There are some exceptions — for example, a reference to a static global variable will always resolve to that definition — but all symbolic references between object files may be assumed to be potentially imported. Therefore, even if fixed addresses instead of PIC are used to achieve pure internal references, compiler changes may be required so that imported references will be made indirectly through the linkage table. Since these references may be slower than ordinary references, it may still desirable to allow the library developer to specify which references are importable and which are not, but this specification should not be made too difficult. For SVR3, AT&T established a convention in which header files were used to reduce or eliminate changes to the body of the source code, but this was awkward. A compiler directive would be cleaner.

For the dynamic linker to resolve symbolic references within a linkage table at run time, each object must contain an *export table* as well as an import table. The export table is usually hashed so that it may be searched quickly. Like the import table, it may be combined with the standard name list for the object, or included as a separate table within the text segment.

When linking against archive libraries, all global symbols are available to all objects, so to achieve complete compatibility with archive libraries, all global symbols defined by an object should be exported. With a few exceptions such as **matherr()**, however, most symbols defined by a program are not meant to be imported by libraries. To reduce the size of the export table in a program, one could export only those symbols that are imported by a shared library. If complete library information is available at static link time, the static linker can detect which user symbols are imported by a shared library, and export only those. Otherwise, an application developer could explicitly state at link time which symbols he expects to be imported by a shared library. Similarly, some symbols defined within a shared library may not be intended for export. The library developer could likewise specify which symbols he wishes to export.

User Program Imports

Recall that in the Multics organizational model, there is no distinction between a program and a library. Multics runs on a segmented architecture on which all addresses are expressed relative to a base register, so internal references are always pure. Imported references are always made through linkage tables. In most other organizational models, however, there is a clear distinction between a program and a shared library. Since traditional UNIX has no shared libraries, changes in coding or code generation techniques to generate shared libraries may be considered acceptable. If possible, however, it would be desirable to be able use existing object code for user programs. There may be existing object files for which the source has been lost, or assembly language sources for which it would be too troublesome to convert to use PIC or linkage tables. Also, since these special coding techniques generally result in slower code, requiring them for user programs could adversely affect performance too greatly.[2]

Since user programs are almost always linked at a fixed address (usually zero), and it is known at static link time exactly which references are imported, special techniques may be used by the linker to guarantee pure external references without requiring special coding techniques. Procedure references can be handled easily, by resolving the call directly to a linkage table entry for the symbol, which is placed at a known fixed address. Data references require the absolute address of the data itself to be known statically. If libraries are at permanently fixed addresses, and relative offsets of symbols within each library are fixed, the references can simply be resolved to the actual addresses of the data.

As observed earlier, requiring library addresses and symbol offsets to remain constant may not be a reasonable restriction. Instead, the linker can simply copy the data into the program, just as it would with data defined in an archive library. This solution defeats sharing of the disk image of data to which the program directly refers, but program data imports are rare in comparison with procedure imports, so the duplication may not greatly reduce the disk space savings shared libraries make possible. Data normally cannot be shared in memory in any case.

Some complications may arise if this approach is taken. The SunOS implementation requires the library developer to copy all exported data into a separate archive file, so that the linker can easily copy the referenced data into the program. All data copied would have to be exported from the program so that the shared library can access the program copy at run time. Aside from placing a burden on library developers, this method is not effective for data items initialized to addresses of other items. To work at all, the items to which the data points would have to be copied into the separate archive file, and then be copied again into the program. If the items are static variables or procedures, then copying them into the program will not work properly. The shared library code may wish to refer to those items as well, but because of the semantics of C, the library will reference its own copy of those items. For this to work correctly, everything that referenced the static items would have to be copied into the separate archive as well, and then would have to be copied again into the program.

A better implementation of the copying idea would be to have the static linker allocate space for the data in the program but not necessarily copy the

[2] Data from the Softbench project suggests that some programs spend more time executing library code than they do executing their own code. For these programs, the use of PIC or linkage tables in library code may also represent a performance problem.

data. No separate archive should be required; the size of each exported data item should be recorded in shared library itself so that the linker knows how much space to allocate in the program. A flag could be set in the program to inform the dynamic linker at run time to look up each imported data symbol in the shared library that originally defined the symbol and propagate the contents of the item to the program. Data that is initialized to absolute constants rather than to addresses could be copied into the program statically, to reduce the expense of copying data at run time.

This method would reduce the amount of data that must be duplicated, since data that is referenced only indirectly would not copied. If library developers are made aware of the mechanisms, they can help further reduce the amount copied by exporting only pointers to large data structures, instead of the data items themselves. There are still some problems with data copying that will be revisited when version control is discussed.

Non Pure Text Segments

There is a way for a program to import shared library symbols that does not require code changes, linkage tables, or copying data, if one is willing to sacrifice read only sharable program text segments. If the pure text requirement for programs is relaxed, then the dynamic linker can resolve the absolute references at run time. The dynamic linking may take longer than it would for programs with linkage tables, since all references to each imported symbol must be resolved, instead of just one linkage table entry per symbol. For small programs that require little relocation, or large programs for which startup cost is not a big concern, this may be acceptable. This method is used in the HP Softbench implementation. The gain from sharing library code should outweigh the loss from not sharing application code in memory. Certain applications such as the shells may be exceptions to this rule. These programs should use linkage tables or be linked with archive libraries so they will still be sharable.

With some loader modifications, however, even an absolute code program text segment that has been dynamically modified can still be shared between processes, if all users of the program are using the same shared libraries, and the shared libraries are loaded at the same addresses in each process. When a program is first loaded, the dynamic linker is invoked to resolve all external references. The loader then marks the text segment read only, and enters the name of the program in a table of active dynamically linked programs. If another process tries to load the program, the loader checks the table and may allow the new process to use the memory image that has already been dynamically linked.

This technique can also be applied to implement efficient global installed libraries. The text segments of such libraries must be pure, so they will automatically be sharable. The dynamic linker is invoked when the library is first installed to resolve all symbolic references for the linkage table, based on a set of default definitions. Processes that provide only default definitions for these symbols will be able to share a copy of the library as well as its linkage table — once the linkages have been established for the library, there is no need to reestablish them for other well behaved processes. Processes that interpose other definitions for symbols imported by the installed library may still share the library, although they will pay the normal price of dynamically linking its imported references. This technique is sometimes known as ''rocket launching'' [7].

Symbol Binding Time

Unless fixed addresses are used for shared libraries on the system, a dynamic link phase is needed to resolve symbolic references at run time. The binding of a linkage table entry can be performed at startup time, or it may be deferred until the symbol is referenced. The latter may be accomplished for function calls by initializing the linkage table entries to point to an entry point within the dynamic linker, since the procedure section of the linkage table is implemented as a jump table. The dynamic linker would have to determine from which linkage table entry the reference came and to what symbol the entry should be bound. It would then resolve the reference, patch the original linkage table entry so that subsequent calls will jump directly to the target, and transfer control to the new routine. This approach is not usually possible for data references, since they cannot be trapped without architecture support, as is provided on Multics. Therefore all data references within a library normally must be bound when the library is first entered.

There are tradeoffs between immediate and deferred binding. Binding all symbols immediately may increase process startup time dramatically. Many symbols that are never used may be bound. Once the binding is complete, however, the process may continue execution normally. If deferred binding is used, startup cost is not as bad, but interactive response time may be slowed as the dynamic linker is invoked over the course of execution. There is also the possibility that, because of a programmer or system administrator error, a required symbol cannot be found at run time, in which case a program abort would normally occur. Some programs may leave the system in an inconsistent state if they abort unexpectedly.

Immediate and deferred binding are not incompatible as design alternatives. It is possible to implement a dynamic linker that, depending on the state of a flag, either binds all symbols immediately, or defers procedure binding until first reference. The flag could be contained within the program itself (possibly set by the application developer with a linker option) or in the user's environment, as is done in SVR4. The former is probably more useful, since the appropriate tradeoffs in reliability and in startup time versus response time are probably traits of a program and not of any particular user. In any case, these methods of setting the flag are not incompatible, either.

Binding Granularity

The granularity at which the virtual memory system views your program and the libraries it uses is what determines the amount of sharing that can take place. I now wish to consider the granularity at which the linkers view a library when binding symbols. This will be seen to affect program size and execution speed, as well as the transparency of the shared libraries implementation.

Static Binding Granularity

In traditional UNIX, an archive library is a single file on disk, but it consists of several autonomous object modules. Each module is copied into an application program only if it defines symbols directly or indirectly referenced by the program. Consider a Fortran math library, which often consists mostly of ''wrappers'' around C math library routines. A Fortran program that references only the Fortran square root routine is linked against both the Fortran and C math libraries. The linker copies the Fortran square root routine into the program, and since the module defining that routine references the C square root routine, the latter is also copied into the program. Although the Fortran math library also defines wrappers around other C math library routines such as the trigonometric functions, none of these are copied into the program since they were not referenced. Therefore the C math library definitions are not copied either.

Now suppose the Fortran math library is replaced with a shared library, but the C math library is left in archive form. The static link phase must be able to determine that the C math library definition of the square root routine should be copied into the program, or else it will not be possible for the dynamic linker to resolve the reference from the Fortran math library at run time. If the shared library is implemented as a collection of object files, as in Multics, then the static linker is able to determine, as it can for routines defined in archive libraries, that the Fortran square root routine does not reference any other C math library routines. Therefore the C math library routines will not be copied into the program, even though they are referenced by other modules within the Fortran math library. If the shared library is implemented as a single executable file, as in SunOS, however, the static linker is not able to determine which modules within the shared library import which symbols from the C math library. It is therefore forced to copy into the program all the C math library routines that are referenced from anywhere in the Fortran library. As a result, the program may be larger when linked against a shared version of the Fortran library than when linked against an archive version.

There is also the potential for a difference in behavior between archive and shared libraries. In the previous example, consider now the case where two additional libraries are listed after the C math library. The first is a special hardware dependent graphics library, and the second is a hardware dependent math library that defines versions of trigonometric routines for the use of the graphics library. If the C math libraries versions of the trigonometric routines are copied into the program, then the versions in the special library will be ignored. If the C math library versions are not copied into the program, then the versions in the special library will be used. While this is a contrived example, it demonstrates the potential for differences in behavior due to the granularity of binding.

Therefore, it can be advantageous for the linker to view a shared library with finer granularity than is possible in the SunOS implementation. This can be accomplished for shared libraries that are implemented as single executable files by maintaining a data structure that lists, for each object module within the library, what references it may make.

Dynamic Binding Granularity

Since the time spent in dynamic linking affects performance, it is desirable to have as few linkage table entries to bind as possible. In an organizational model like the one in Multics, in which individual object files are shared separately, the C library may comprise several hundred object files that all reference the global variable **errno**. Each of these files will contain a separate linkage table, each with an entry for **errno**. Hundreds of entries for the same symbol will be resolved at run time for any given program. In an organizational model like the one in SunOS, in which the object files composing a library are combined by the linker, there will be only one linkage table entry for **errno** that needs to be resolved at run time. Therefore it is clearly advantageous to combine object files when possible, to reduce the duplication in the linkage tables.

On the other hand, consider again the example of the Fortran program that calls only the Fortran square root routine from the math libraries. Previously, we observed that implementing a data structure that associated modules with references would prevent unreferenced routines from being copied into the program from an archive version of the C math library. Assume now that both the Fortran and C math libraries are in shared form. At run time, the dynamic linker may have to bind all the references from the Fortran math library into the C math library. Even if deferred binding is used, so that procedures are bound only if referenced, all data references must be bound immediately, since it is not usually possible to trap data references. Therefore the same type of data structure used during the static link phase can be used during dynamic linking to reduce the number of linkage table entries that must be resolved.

Besides resolving the linkage tables, there may be additional dynamic relocation that must be performed by the loader. For example, variables which are initialized to the addresses of other data items will need to be relocated at run time. Normally, a loader must perform all dynamic relocation immediately. If the granularity of binding is improved, then the loader may relocate only those data items that are reachable. This is sometimes known as *lazy relocation*.

Version Control

One advantage of shared libraries is that a library developers may improve a library routine, and application programs can benefit from the change without relinking. The other side of the coin is that a change may cause application programs to break unexpectedly. The goal of version control in this context should be to afford programmers at least as much control as they have with archive libraries. When a programmer changes a routine in an archive library, application programs will only be affected if they are relinked. Ideally, *compatible* changes to a shared library should be seen by applications immediately, while *incompatible* changes should not affect a program until the program is relinked.

Detecting Incompatibility

We have already come across one type of incompatibility, which I call *internal* incompatibility. An example of this is the two versions of **malloc()** that cannot both be used by the same program without corrupting the heap. Version control is usually more concerned with *external* incompatibility, in which there are two versions of a routine that cannot be used interchangeably. For example, one version may take different parameters, return different values, or have different requirements or effects. An externally incompatible change is *backwards* compatible if references to the old version can safely be satisfied by the new. It is *forwards* compatible if calls to the new version can safely be satisfied by the old.

As was the case when discussing the symbol namespace problem, dynamic type checking is not enough to detect externally incompatible changes. For example, there may be a routine called **memcpy()** with the type signature "void *memcpy (void *, const void *, size_t)." The original implementation of this routine may work correctly if the two arguments overlap. If at some future time, the library developer replaces the algorithm with a faster one that does not correctly handle overlap, this would constitute an incompatible change. This change would not be backwards compatible, but it is forwards compatible. Changing from an algorithm that does not handle overlap to one that does would be an example of a backwards compatible change that is not forward compatible. Changing the return value would be neither backwards nor forwards compatible.

Handling Incompatibility

It must be assumed that the library developer will be able to identify externally incompatible changes. It must be the library developer's responsibility to make sure existing programs will not be adversely affected by changes that are not backwards compatible. One solution is to give a different name to the new version of the routine, so that existing programs can continue to call the old version. If a program is linked against the new version, but is run on a system with only the old version, the dynamic linker can prevent the program from being executed.

Changing the name of the routine may not always be a viable option. There may be standards that motivate the change, and the standards may dictate the name of the routine as well. One solution is to simply rename the library. Assuming the name of the library was recorded during the static link phase of the program, the loader would map in the old version of the library, so the change would not affect the application. If a program is linked against the new version, but is run on a system with only the old version, the loader can prevent the program from being executed.

Changing the name of the library may not always be reasonable, either. In SunOS, a convention of version numbers for libraries was established. All versions of a library have the same base name, but a suffix indicating the version number is appended. When a program is linked against a library, the linker records the version number of the highest available version. The loader then loads the same version at run time. Since all versions of the library are kept on the system, the application developer has the option of explicitly relinking with an old version of the library.

This approach has some drawbacks. Duplicating an entire library for an incompatible change to one routine not only wastes disk space, it also prevents programs using two different versions of the library from sharing the library in memory. This scheme could be improved by simply adding *satellites* for incompatible changes, instead of duplicating the entire library. For example, the original C library may contain the version of **memcpy()** that handles overlap correctly. If at some point the library developer wishes to introduce the faster version that does not handle overlap, he may create a separate satellite shared library containing only the new version of **memcpy()**. The original version would remain in the original library. Existing applications that were linked with the C library would not load the satellite, so they will not be affected by the change. The static linker would automatically include all satellites when linking against a library, unless the application developer explicitly states otherwise. At run time, programs linked against the satellites would load them as well as the original library, and would use the satellite versions of changed routines.

The satellite approach increases the level of sharing, but it may fragment libraries over time, as individual routines are placed in separate files. Each of the satellites must be loaded individually. While loading libraries is cheap, it is not free. Also, each satellite contains its own linkage table that will often duplicate entries in the original library. The dynamic linker may need to resolve some symbols several times. There is also the administrative problem of maintaining the satellites.

A solution to these problems is to include the satellites within the library itself. Several versions of a given routine may thus coexist in the same shared library. A mechanism must be introduced to insure that old programs continue to bind to old versions of changed routines, while new programs bind to new versions of changed routines. One way to accomplish this is to associate each version of a routine with a conceptual satellite name. The linker records in the program the names of all the satellites within each library. The dynamic linker at run time would only accept symbol definitions that were associated with one of the satellites present at static link time. Furthermore, to protect programs expecting a newer version of a library from being executed on a system with only an older version, the loader or dynamic linker can prevent a program from being executed if it finds that not all the expected satellites are present within the shared libraries at run time.

So far I have discussed only incompatibilities in procedure calls. It is also possible that a library developer will change the initial contents or size of a data item. This can be a problem if the item had been statically allocated in an executable that referred to the item. The old value may be contained in the program, and the library may be expecting the new value. It will usually be unacceptable to allow the program and the library to access separate copies of the data item. Therefore, it seems that whenever the initial value or size of a library data item changes, an incompatible change has occurred. If any library routines are changed to depend on the new initial value or size of the data item, these also must be considered incompatible changes. Programs using the old value of the data item should be able to continue using the old versions of these routines.

It may not be necessary to identify changes in initial value as incompatible changes. If the size of the data item has not changed, or has become smaller, then the dynamic linker could copy the new value into the program at run time. The program could then access the new initial value of the data item, and could safely use library routines that depend on the new initial value. If the data item becomes larger, then there will not be enough room allocated in the program for the new version of the data item. In this case, the program must continue to use the old value of the data item and the old versions of library routines that depend on this value.

Other Issues

There are several other issues that must be considered when designing a shared library interface for UNIX systems. While there is no room to discuss these in detail, some of them are listed below.

Dynamic Loading

It is often desirable to be able to specify a shared object at run time to be loaded into the process. A user interface should be created to allow a user to load and unload objects, and to refer to symbolic addresses within them.

Transparency

When introducing a shared libraries implementation to a system, it is desirable to make the design as transparent as possible. Users who are accustomed to program development and execution with archive libraries should encounter as few surprises as possible when using shared libraries. The importance in this regard of library binding time, granularity, and version control has already been established. There are other issues to consider as well. If the dynamic linking functionality is provided in user space, it should be careful to avoid the use of **malloc()**, which may disrupt the application's own memory allocation. Similarly, UNIX programs occasionally depend on the stack segment being initialized to zeroes on invocation of **main()**. Although neither C nor UNIX guarantees this behavior, most implementations provide it. A user space dynamic linker invoked before **main()** may violate this assumption, however. Also, errors that occur during loading and dynamic linking should be handled gracefully. Debuggers should be enhanced to understand shared libraries.

HP-UX Shared Libraries Implementation

We have implemented shared libraries on HP-UX for both the Series 300 (680x0 based) and Series 800 (RISC based) systems. Although there were several previous implementations on HP platforms (MPE, HP-UX on the Series 500, and the Softbench product for the Series 300 and Series 800), these designs were deemed unsatisfactory since they did not adequately address the concerns raised in the preceding sections. Our design is primarily based on the SunOS and SVR4 schemes, extended to resolve some of the deficiencies noted previously. The descriptions below are based on the Series 300 implementation. The Series 800 implementation is similar, although it does not yet implement the improvements we made to binding granularity.

Overview

Shared libraries on HP-UX are created from object files containing position independent code. All references to global symbols are made through a linkage table, and all global symbol definitions are exported. The object files combined by the linker to form a file in the same format as an executable file. When a program is linked against a shared library, the full pathname of the library is recorded in the executable file, as opposed to a filename only as in SunOS and SVR4. The linker creates linkage table entries for program references to shared library procedures, and copies shared library data items referenced by the program directly from the shared library into the executable file. Data items initialized to point to other shared library symbols are specially marked so the dynamic linker can recalculate the addresses at run time. No separate archive file is used as in SunOS, nor are items to which copied data points copied into the program. Only symbols imported by a shared library are exported from a program, although all symbols may optionally be exported. A linker flag sets a bit in the executable file to control whether immediate or deferred symbol binding should be performed by the dynamic linker. No run time flag is used as in SVR4.

At run time, the loader maps the program into memory and transfers control to the program entry point. The startup code maps in a dynamic loader and calls it to load the shared libraries. The dynamic loader maps each library into memory at addresses unlikely to conflict with the program text, data, bss, shared memory, or stack segments. The dynamic loader also performs all dynamic linking functionality. If deferred binding is specified, then the entries in the procedure section of the linkage tables are initialized to jump to the "bind on reference" routine within the dynamic loader. For each specially marked data item in the program, the dynamic loader finds the copy in the original shared library and propagates its value back to the program. If immediate binding is specified, all symbolic references are bounded immediately.

When the dynamic loader has finished its startup tasks, it returns to the program startup code, which invokes **main()**. If immediate binding was specified, execution continues uninterrupted until the process exits, at which point the system automatically unmaps all shared libraries. If deferred binding was specified, when a library function is called for the first time, the linkage table entry transfers control to the dynamic loader, which determines the source and target of the call. The dynamic loader must also determine which data items may be referenced by the target routine, and bind the linkage table entries for those as well. The dynamic loader then patches the original linkage table entry, and transfers control to the target routine.

This is basically the same approach used in SunOS and SVR4, with minor differences as noted. Note that since absolute pathnames are used, it would appear this design does not allow enough flexibility for the system administrator. This should not be a problem, however, since HP-UX supports the concept of a *context dependent file* (CDF). Context dependent files provide an invisible directory level and allow several physical versions of a file to be associated with the same pathname, with the *context* of a process determining which will be accessible. Thus in a mixed cluster, both Series 300 and Series 800 versions of the C library may coexist as **/lib/libc.sl**. Processes executing on a Series 300 client will access the Series 300 version, while processes executing on a Series 800 client will access the Series 800 version.

The biggest difference between the HP-UX design and the SunOS and SVR4 designs is in the area of version control. The Series 300 also implements fine static and dynamic binding granularity for shared libraries.

Version Control

The version control scheme designed for HP-UX uses the idea of satellite libraries that are included within the library itself. Instead of recording names for the satellites, however, we use the concept of a *high water mark* to identify incompatible changes to a library. All incompatible versions of a routine are included in the shared library. Each version after the original is stamped via a compiler directive with the date (month/year) the change was made. When a shared library is created, the linker records in the header the date of the most recent change — the high water mark for the library. When a program is linked against the shared library, the high water mark for the library is recorded in the executable file along with the pathname of the library. If the program is run immediately, the dynamic loader will resolve all symbolic references to the most recent definition of each symbol. This is consistent with the behavior of archive libraries, where only the most recent definition of any given symbol is present within the library.

If a subsequent incompatible change is made to a library routine, it is also included in the library. The date stamp of the new version of the routine will be more recent than the high water mark of the previous version of the library. If several routines are changed at once, all may be given the same date stamp. The new version of the library will contain all versions of the changed routines. The new high water mark of library will be equal to the date stamp on the new versions of the changed routines. This will therefore be more recent than the high water mark of the old version of the library. The new version of the library then replaces the old version.

At run time, the dynamic loader will refuse to load a library if its high water mark is older than the mark recorded for the library when the program was statically linked. This protects a program from being run with a version of a library that is too old, since the old library may not contain the required versions of some routines. The dynamic loader will accept a library only if its high water mark is greater than or equal to the mark recorded for the library when the program was statically linked.

When binding a symbol, the dynamic loader will not accept any definitions that are more recent than the high water mark recorded for the library when the program was statically linked. Therefore the program is not affected by any incompatible changes made since the program was initially linked. The dynamic loader will resolve a reference to the most recent definition that is less than or equal to the high water mark recorded for the library when the program was statically linked. Since the library would not have been loaded if it was too old, this guarantees that each reference will resolve to the same definition as it did when the program was initially linked.

When the library developer makes an externally incompatible change that is backwards compatible with the previous version, he need not include both versions of the routine, nor should he change the date stamp. Instead, he should merely include in the library a dummy module that contains the date of the change, to force a change in the high water mark of the library. This way, the dynamic loader will allow programs that were initially linked with the old version to use the new one, because the date stamp on the routine has not changed. The dynamic loader will not allow programs that expect the new version to load the old library, because the high water mark of the library will be too low.

Binding Granularity

Each export table entry in a Series 300 shared library contains an index into a *module import table.* The latter table contains an entry for each object module. Each entry contains a list of symbols referenced by the module. This enables the linker to handle correctly cases like the Fortran square root program discussed earlier. If a program is linked against a shared version of the Fortran math library and an archive version of the C math library, and the program contains references only to the Fortran square root routine, then the Series 300 linker will mark the C square root routine as undefined when the definition of the Fortran square root routine is encountered, since the C square root routine is called by the Fortran square root routine. The C square root routine will then be copied into the program, and it will be exported so that it can be used at run time to satisfy the reference from the Fortran math library. It will not mark as undefined any of the other C math library routines imported by the Fortran math library, so the other routines will not be copied into the program. Without the module import table, the linker would have to mark all imported symbols as undefined to cause the C square root routine to be copied in from the C math library. This would cause all the other C math routines to be copied as well.

If immediate symbol binding is used, a dynamic linker must normally bind all linkage table entries in all libraries. This can be extremely time consuming, particularly in programs that use many libraries. The Series 300 dynamic loader is able to use the module import table within the shared library to determine which references are reachable. It binds only those references that are encountered in a depth first search starting with the symbols imported by the program itself.

When deferred binding is used, a dynamic linker must normally bind all data references within a library the first time any procedure in that library is called. The Series 300 dynamic loader is able to use the module import table to determine which data references are reachable. When resolving a procedure call, it binds only those data references that are made by the module defining the called procedure.

Results

The advantages and disadvantages of using shared libraries are already well known. Typical results show about 45% to 60% savings in executable file size, with anywhere from 0% to 10% performance degradation, on typical UNIX programs built with shared libraries. Reductions by a factor of ten or more in the size of X window programs are common. The disk space savings are attained because library code is not copied. The performance degradation is caused by a combination of position independent code and dynamic linking. Performance is generally markedly worse on short programs dominated by startup cost, often by factors of several times on trivial programs

The implementation on the Series 300 yields similar characteristics. Features such as copying only directly referenced shared library data into a program and exporting only those program symbols that are imported by a shared library helped us to achieve an additional 10% to 20% size savings. The cost of position independent code on the 68030 is about 10%. Since it is used only in the library code, this has a negligible effect on the performance of most programs.

Most of the performance degradation caused by shared libraries can thus be attributed to loading the libraries and dynamic linking. For programs dominated by startup costs, such as a trivial "hello, world" program, the library loading cost can almost double overall run time. The dynamic linking cost would be over 50% of the original run time, but because the Series 300 dynamic linker binds only those symbols that are reachable, we were able to cut the dynamic linking cost by a factor of five (to 10% of the original run time).

For programs not dominated by startup costs, the library loading cost is usually insignificant. The dynamic linking cost remains about the same for non trivial but short programs (several seconds of execution time). There is not as much advantage to be gained by binding only reachable symbols in longer programs, and this advantage approaches zero as the number of symbols referenced approaches the total number of symbols defined. For typical UNIX programs such as *ld*, the advantage is reduced from a factor of five to a factor of two. The overall performance degradation due to shared libraries for such programs is around 5%.

Immediate binding generally increases dynamic linking costs, since symbols that are never referenced may be bound. The amount of increase varies from under 10% in programs that use only the C library to factors of three or more in X window programs.

More important than the numbers, for many people, is the usability of the system. One of our prime concerns was to achieve as high a degree of transparency as possible. We constrained our design to require no source changes to build either shared libraries or programs, and to eliminate the need for a separate archive library for shared library data. Features such as the fine static binding granularity of shared libraries helped us to achieve almost complete compatibility with the behavior of archive libraries in most situations. Our version control scheme provides a simple yet effective means to protect programs from incompatible changes.

There are some areas in our implementation with which we are not entirely satisfied. We do not support true source level debugging within shared libraries at all, although a person debugging a program that uses a shared library can view or single step through symbolic disassembly of the library. Programs in our implementation use fixed pathnames to locate shared libraries at load time, although this is not as serious a limitation as it would be on a system that did not support context dependent files. We now understand how to provide load time search paths and user installed libraries without causing security problems, so these features may be added at a later release.

In general, however, we were able to address almost all the issues discussed in this paper, and resolve them to our satisfaction.

Acknowledgements

I would like to recognize the members of the HP-UX shared libraries design and development teams — Stacy Martelli, Cary Coutant, Michelle Ruscetta, Brad Stallsmith, Bob Schneider, Carl Burch, Edgar Circenis, Steve Lilker, and Greg Lindhorst.

References

1. AT&T, Application Development Environment, *Software Developer Conference*, 1989.
2. AT&T, *UNIX System V Release 3.0 Programmer's Guide*, Prentice Hall, Englewood Cliffs, NJ.
3. J. Q. Arnold, Shared Libraries on UNIX System V, *USENIX Summer Conference Proceedings*, 1986, 1-10.
4. Hewlett-Packard Company, HP Link Editor/XL Reference Manual, 1987.
5. Apollo Computer, Software Development on Domain/OS, 1988.
6. R. C. Daley and J. B. Dennis, Virtual Memory, Processes, and Sharing in MULTICS, *Comm. of the ACM 11*, 5 (May 1968), 306-312.
7. Open Software Foundation, Lecture 3, *Mach Lecture Series*, Nov. 1989.
8. Open Software Foundation, OSF Loader Functional Outline, internal design document, 1990.
9. R. A. Gingell, M. Lee, X. T. Dang and M. S. Weeks, Shared Libraries in SunOS, *USENIX Summer Conference Proceedings*, 1987, 131-145.
10. J. Hobbs, Installed Shareable Images, *Dec. Professional 6*, 4 (Apr. 1987), 78-82.
11. G. Letwin, Dynamic Linking in OS/2, *Byte 13*, 4 (Apr. 1988), 273-80.
12. S. Sands, Personal communication.
13. M. Solinski and M. Johnson, Dynamic Link Libraries Under Microsoft Windows, *Dr. Dobb's Journal Of Software Tools 14*, 3 (Mar. 1989), 28-9,32,35,37,82-91.

Marc Sabatella received his BS in Computer Science and Mathematics from Florida State University in 1987 and his MS in Computer Science from the University Of California at Berkeley in 1988. Since then, he has worked in the Hewlett-Packard Colorado Language Lab, where he participated in the design of shared libraries for HP-UX, in cooperation with members of several other labs. He coordinated the compiler and linker changes and implemented the dynamic loader for the Series 300 workstations.

USENIX Summer Conference
June 11-15, 1990
Anaheim, California

Shared Libraries as Objects

Donn Seeley – University of Utah

ABSTRACT

The C++ programming language can provide a flexible and efficient model for sharing code and data among processes. Like a library, a class defines a set of symbols to reference exported code and data; like a library, a class permits one to override basic functions with new code (a virtual function). This interposition is a form of dynamic linking, since it permits references to be substituted at run-time, although it does not look up names in a symbol table at run-time. This paper describes an implementation of shared libraries using the GNU C compiler on Berkeley UNIX which mimics C++ classes and provides position-independent code with dynamic linking at considerably less expense than more traditional dynamic linking schemes.

Introduction

This paper is about an implementation of *shared libraries* for Berkeley UNIX [Leffler89]. Shared libraries differ from ordinary libraries in that their text and sometimes their data are incorporated into a process at run-time, and are shared among different processes. This run-time loading and sharing can result in considerable savings of disk space and memory across many binary files and many processes. The implementation of shared libraries described in this paper differs from more typical implementations such as SunOS 4.0 [Gingell87] in that it is motivated by *object-oriented* programming concepts. The term 'object-oriented' means different things to different people [Stroustrup87], but in this paper I will use it to refer to a programming style in which operations center around 'types' (named collections of data structures and functions that operate on them), such that new types can be built by extending old ones in a well-defined way. An 'object' of such a user-defined type provides an encapsulation of a resource that can be used without knowledge of the internal structure or state of the resource. Object-oriented programming promotes sharing through inheritance, and flexible re-use of existing software is a major advantage of this approach; the closest that the old technology came to this flexibility was the library. An implementation of libraries that adds new capabilities for sharing and inheritance enhances old software and creates new possibilities.

This paper will start out by motivating a new shared libraries implementation and explaining the background to the approach we took. Then we'll get into the nitty gritty of the current implementation, explaining what's good and what needs to be improved, following that up with a few statistics on the effectiveness of sharing and the run-time performance overhead. In the last part, we'll lay out our some of our plans for extending this scheme into uncharted regions.

Motivation

In the past year or so, shared libraries for Berkeley UNIX have become both technically feasible and operationally necessary. The biggest technical advance has been in the reliability and availability of the GNU C compiler from the Free Software Foundation [Stallman90]. GCC is a well-structured, retargetable, highly optimizing C compiler for which source code will always be available, by condition of the license. GCC has now been ported to a wide variety of architectures, including the VAX, Motorola 68000 and 88000 series, MIPS, Intel 386 and 860, SPARC and others. Its reliability is now so good that the Utah/mt Xinu distribution of 4.3 BSD for the 68000-based Hewlett Packard 9000 series 300 machines uses GCC for its main C compiler to compile all of UNIX; perhaps more notably, NeXT uses GCC to build all of its system software. GCC is the obvious choice for the compiler for any freely available implementation of shared libraries based on position-independent code.

Another essential technical advance is a new version of Berkeley UNIX, which finally contains file mapping memory management. The code for this will be freely available, deriving from the Mach OS from CMU. The initial port of this new virtual memory system to UNIX was done here at Utah on HP machines, and it has formed the platform for the shared libraries implementation.

At the same time that the technical pieces were coming together, the practical motivation for using shared libraries became stronger. Libraries have been growing in size — only a few years ago, the code and data from a library that were included in an application by the linker typically might fit in a page on a machine like our HPs, but with the advent of features like graphics interfaces such as X, shadow passwords and nameservers, this is no longer the case for many binaries. Network file service can spend a significant amount of time shipping pages of binaries that contain library code which could instead be shared by in-core applications on a client; shared libraries sometimes reduce binary images to a fraction of their old size by eliminating redundant information, and this improves throughput. Given our experience with compiler support under Berkeley UNIX and our interest in shared libraries, we felt that our technical capabilities and our pressing need for sharing were excellent motivation for proceeding with a freely available implementation.

Background

This section briefly introduces basic concepts of shared libraries and C++ objects, then explains how they can fit together.

Shared Libraries

I'm going to define several terms, partly to introduce them to readers who aren't familiar with them, and partly to indicate my own idiosyncrasies. Readers who already understand the concept of shared libraries can skip ahead.

The simplest aspect of library use is *loading.* A process loads a shared library by bringing it into its address space, for example by mapping the file that contains the library. Loading is often confused with *linking,* which is the process of resolving symbolic references into references to particular addresses within code or data. A compiler or assembler will convert a call to a function named *open* into a code template that must be filled in by a linker; the linker will examine the symbols defined by a library and if it finds a definition for *open,* it will patch the call to refer to the code for the function. *Interposition* can occur when the definition for a symbol is not unique; the linker will ordinarily choose one definition and ignore the others, interposing the first definition before the remaining ones. The idea of interposition becomes useful when you want to modify the behavior of a library — for example, you could arrange for the linker to scan a library containing your special version of *open* before it checks the system library, and your version would be used in preference to the standard one.[1]

Implementations of shared libraries usually distinguish between *static linking* and *dynamic linking,* which refer to linking before run-time and at run-time, respectively. Some very rudimentary shared library implementations use only static linking; the system assigns a library to a fixed address range in every process and thus it is possible to predict prior to run-time the locations of library code and data and link to them. Sometimes a static linking implementation will use a *function dispatch table,* an array of pointers to functions in the library, so that there are no linking dependencies on the code sizes of functions. Fancier implementations use both static and dynamic linking; static linking is performed by a *batch linker* (this is a retronym for the system's traditional linker/loader) and dynamic linking is accomplished using a run-time linker that is either always resident or is bootstrapped by a mini-loader in the standard start-up code for a program. UNIX System V Release 3 used only static linking [Arnold86], while System V Release 4 has dynamic linking.

To make run-time loading efficient, it is useful for the compiler and assembler to generate *position-independent code* or *PIC.* PIC is designed to reduce run-time *relocations,* which are changes made to data, or sometimes code, to indicate the location of other code or data in the address space. A statically initialized pointer is an example of data that must be relocated — after a process loads the object that the pointer points to, it must patch the pointer to indicate the object's absolute address. Lists of addresses to be patched are kept in a *relocation table.* PIC reduces relocations by arranging for the code itself to compute the proper address when it needs it. In *PC-relative* addressing, the code adds the program counter and an offset to generate a pointer; when the code is at a fixed offset from the address it needs to reference, this strategy will work no matter where the code is loaded. *Base pointer* addressing is more general than PC-relative addressing — code and data are accessed by adding fixed offsets to the value of a base pointer. The base pointer often resides in a dedicated register (a 'base register') but may also be loaded into a register on demand, using PC-relative addressing or another PIC strategy. Base pointer addressing permits code to access other code and data which are not at a fixed offset from the current PC, which PC-relative addressing cannot do directly.

Dynamic linking and run-time relocation can be expensive because they can lead to virtual memory overhead. When a process writes into a shared library page for the first time, the system must make a private copy of the shared page; this is called a *copy on write* or *COW* fault. These faults are expensive in two ways: they defeat sharing, and they use up cycles. One way to improve sharing is to arrange for the compiler to generate *pure text,* moving relocated pointers out of compiled code into a table and changing references to go indirectly through the table. Run-time relocation will modify the table but leave the executable code unchanged and sharable. One way to decrease cycles is to improve *locality of reference,* so that fewer pages are faulted; for run-time relocations, this involves storing relocated values close together so that fewer pages fault with COW during relocation.

[1] Rob Gingell of Sun and I independently came up with a lot of exotic ideas for using run-time interposition. He has done much more with it than I have, and I have therefore liberally borrowed from him.

C++ Objects

This section will contain a short review of C++ [Stroustrup86] and its style of object-oriented programming. The reader should skip this if they are already familiar with C++ implementations. I will use several terms from object oriented programming in a way that will sound heavily biased toward C++; I don't wish to imply that C++ is qualitatively better in some way than other object oriented programming systems, but later material will refer to C++ and its implementation, and this section is intended to provide context.

Object oriented programming in C++ centers around two concepts, *data abstraction* and *inheritance*. C++ promotes data abstraction through user-defined types called *classes* that encapsulate the state of a resource. An *object* is an instance of something defined as having a particular type. Unlike a C structure, a C++ class consists of declarations for both data and functions designed to operate on the data; C++ calls functions that belong to a class *member functions*. A class may distinguish *public* and *private* members; private members are accessible only to member functions of the class, while public members can be used by any code that creates an object of the appropriate type. Public members serve as the interface between the resource managed by a class and clients of the resource; just as a C source file must have a structure in scope in order use it, a C++ source file must include a class declaration at compile time in order to access member names and types.

One of the goals of object oriented programming is to promote sharing. Inheritance achieves this by permitting users to define new types that incorporate old ones. A class declaration may specify one or more *base classes* to inherit from; this *derived class* will contain all the members of its base classes as well as any new members. (Note that the derived class is restricted from referring to private members of base classes just like any other user of those types.) A base class may declare a member function to be eligible for redefinition in a derived class; such a function is called a *virtual function,* and the feature provides great flexibility in modifying the behavior of a base class. For example, a base class with a virtual function called *open* that opens a file may be used by a derived class with another *open* function that (say) searches a path for a file before opening it.

C++ permits a pointer to a (public) base class to refer to an object of a derived class. If the derived class redefines a virtual function in the base class, the code generated by the C++ compiler must be prepared to select either the base class function or the derived class function depending upon the actual type of the object at run-time. This means that virtual functions require a special, alternate linkage that uses an indirection; since the actual function that will be executed isn't determined until run-time, virtual functions represent a curious form of dynamic linking. C++ implementations typically hide a pointer inside an object to make the needed indirection; the pointer points at a *virtual function table* or *VFT,* which contains an array of function pointers, one for each virtual function defined in the class. To call a virtual function *open* given a pointer to an object, the compiler converts the name *open* into a constant offset, then generates code to fetch the VFT pointer from the object and call indirectly through the function pointer found at the given offset in the VFT.

A C++ class may define a *constructor*, which is a function that is called automatically when an object is created and arranges to allocate memory and to initialize members. A C++ implementation must provide a kind of implicit constructor that allocates memory in a context-dependent fashion and provides a pointer to this memory to the user's constructor which it may use or ignore as desired; this implicit constructor also calls any constructors for the current class's base classes. A class may *overload* its constructor by providing different constructors for different initializer types or counts. For example, a class may have one constructor which accepts a string as an argument and another that accepts a string and a count; the second form could be used to indicate that the constructor should use the count to extract a substring from the string initializer. A class may also have a *destructor* to clean up after an object when it is deallocated.

Shared Libraries as Objects

Our basic premise is that C++ objects can be a model for an implementation of shared libraries that can be used with ordinary C programs under UNIX. (Of course, these same ideas could be used to create a rich C++ environment under UNIX; more about that later.) Here are some of the analogies that we can draw between shared libraries and C++ objects that have guided our implementation strategy.

Public vs. private code and data: A library has certain external symbols, principally names of functions and data items, while other symbols are static and unavailable for linking from the outside. This level of abstraction is all that is available from C, but it can be subsumed nicely by the public / private scoping of C++. A library's collection of external symbols can be considered its public interface, with each function and data element representing a 'member' of the library. There is still one big difference between public class scoping and external library scoping, which is that a class scope is named; ordinarily external symbols are not uniquely marked as coming from any particular library. There is nothing intrinsic to the name *open* that tells a compiler that it should resolve it in the C library.

C++ class scopes are not entirely free of this problem either, as it turns out. In C++, a class declaration establishes a special scope that applies to member functions, in which unqualified class member names are automatically scoped to the class. For example, a member function *open* of class *libc* has a fully qualified name of *libc::open,* but in the code for a member function of *libc,* it may be called with just the name *open.* The potential for ambiguity arises when a derived class has more than one base class that declares a member of a particular name. Suppose a derived class has base classes *libc* and *libx,* both of which declare a member function *open.* When a member function uses the name *open,* what happens? In C++, the compiler prints an error saying that it cannot resolve the name to a single function. This is analogous to a 'multiply defined' error from a batch linker that finds it must load two definitions for the same name. C++ provides a way out of the ambiguity by introducing the '::' scoping operator so that a reference to a name can be fully qualified. For shared libraries that can be accessed from C, we can treat library members as though they were implicitly scoped to library names, and provide a way to resolve ambiguities using some kind of scoping escape; for example, *SCOPE(libc, open)* might be a way of writing *libc::open* in C. Unresolved ambiguities can be errors as they are in C++, or the compiler can handle them with the traditional semantics of interposition — the compiler's source of library names can also indicate a link order that determines interposition.

Virtual functions and interposition: To pursue the issue of interposition a bit further, let's ask what happens when instead of having a clash between two unrelated functions with the same name, there is a need to develop a new function to replace some standard existing function. In C++, we can declare a new derived class which re-defines a virtual function in the base class. With UNIX libraries, we could define the function in a new library, and arrange to link that library before the standard one; that is, we could use interposition. If we implement shared libraries as objects, we can interpose a function at run-time in the same manner as a C++ virtual function. When an application calls a function in a shared library, it will branch indirectly through a dispatch table. To achieve interposition, the implementation substitutes a modified table that replaces entries for re-defined functions and fills in the rest so that they point at the base functions. This table serves exactly the same purpose as a C++ VFT: it allows the implementation to defer part of its linking to execution time. If we build a shared library with base pointer PIC such that one base pointer points at its functions and another points at its function dispatch table, table substitution is trivial and has precisely the right effect.

Constructors: Another analogy between shared libraries and C++ objects lies in the area of initialization. C++ objects are initialized at run-time by constructors. Shared libraries also have to do initializations, some of which are general and some of which may be library-dependent. The most obvious initialization is relocation. We can think of initialized pointers as having a type that requires construction. In fact, it is not very difficult to write a C++ class that encapsulates pointers and relocates initialized pointers at run-time; the class should have an overloaded constructor, with an empty function for the constructor with no arguments, and a function that takes a pointer argument and relocates that argument for the initialization case. A typical C++ implementation will not generate any code for uninitialized objects of this class, while it will perform run-time relocation on initialized objects. Since relocation is so common and is an initialization that must be done automatically by the shared library implementation, this explicit constructor model can't apply directly, but it seems reasonable to make relocation one of the features of the standard implicit constructor for a shared library.

There are more subtle uses of constructors in shared libraries. One shared library may require the services of yet another shared library; a shared library constructor would be responsible for mapping any other required libraries and in turn calling their constructors. A derived library will also need to map its base library and call its constructor, in a manner analogous to derived class constructors calling base class constructors, and it will additionally initialize its function dispatch table. (The difference between these two cases is that in the first case, one shared library is merely a client of the other, while in the second case, one shared library is extending the semantics of another.) A user-defined constructor for a shared library can do important work. For example, a derived library might trace calls to functions in a base library. The constructor for the derived library would open the trace file and allocate a trace buffer. A destructor for the library would also be required, to flush the buffer.

Inheritance and sharing: Sharing is one of the basic goals of object oriented programming and the main purpose of shared libraries, and under the model we are developing here it is not very difficult to undertake both styles of sharing at once. Inheritance is conceptual sharing: a derived class obtains function and data members from its base classes. Shared libraries represent concrete sharing of binary code and data across processes. One of the benefits of inheritance is that an improvement in the implementation of a base class is reflected in all the classes that derive from it. Similarly, with shared libraries, an improvement in the implementation of a library is reflected in all the clients of that library, without the need to recompile. Moreover, under the model of inheritance we are using for libraries, all the instances of a base library, in the same or different processes, share the same pages (modulo COW faults). Base classes can be shared physically as well as conceptually.

Compile-time pre-linking: One of the reasons why C++ code is typically faster than code from other object-oriented languages is that a C++ implementation can do much of its work at compile time instead of at run time. We can pre-link virtual functions at compile time by converting a function name into a pointer to a table and a constant offset into the table that is unique to that name within that function's class. At run time, jumping to the proper code is a matter of two indirections. On the other hand, classic dynamic linking schemes for shared libraries postpone much of their work until run time, and this creates a run-time system that is often complex and expensive. Since the dynamic linker is a critical path for almost every program, considerable effort must be expended on special hacks which improve the speed of symbol look-up and keep the size of symbol tables down. Some dynamic linking implementations have the capability of checking argument counts and types at run time as well, for even greater run-time overhead. Typically the symbol table format for the dynamic linker is different from that used by the batch linker, resulting in the paradox of two symbol tables for each binary file and two linker implementations to maintain.

A shared library implementation that takes its cue from C++ can be radically simpler and substantially faster than a classic dynamic linking implementation without sacrificing two of the latter's most valuable features: position independence and interposition. The compiler converts a function name into a base pointer and an offset into a dispatch table, much like a C++ VFT. At run time the program uses the address in the dispatch table at the given offset to locate the code for the proper function. The binary files for the application and the library contain no symbol tables, and thus no dynamic linker is needed to read them; the name of the library and the dispatch table offset are sufficient to encode the identity of the function.

Despite the virtues of this approach, there are still a few obvious problems to overcome. One big difference between C++ objects and UNIX libraries is that a class definition for an object must be in scope when compiling something that uses it. As we have seen, the members of a shared library are the functions and data that are declared with external scope; unless these have been carefully declared somewhere (e.g. in a C include file with type declarations and prototypes), this information is not easily available until after the library has been compiled, so how do you compile the library? There are at least a couple ways out of this dilemma: the library can be compiled twice, once as a normal library to extract symbols and establish a mapping of names to offsets, and then subsequently as a shared library using base pointer PIC with the offsets; or, the library will be compiled once and the batch linker can fix references in the binary files after determining the scope and hence the linkage of the various symbols. In either case, the final mapping of names to offsets must be the same for every compilation or linking stage that needs to link with a particular shared library; this is analogous to the requirement that every C++ compilation unit in a program must incorporate the same class definitions in order to use objects of those classes. All this is inconvenient, but not too arduous to implement. The problem becomes more difficult when we consider the issue of compatibility. If we change a library so that we alter the mapping of names to offsets, it becomes incompatible with all the applications that we compiled before. A special case of this occurs when two sites with the same architecture build different library mappings — they may no longer trade executable binary files that use the shared libraries involved. We can alleviate this problem if we can move the mapping from site to site with the library, but it remains a problem.

In spite of the problems in propagating compatibility, we would argue that there are virtues of consistency and type safety in having a standardized interface to a library. An implementation that understands about a library configuration at compile time can check data types, return types, argument types and argument counts at compile time, when that information is most valuable. Given that interfaces do sometimes change, however, an advantage of linking at compile time is that interfaces can evolve by extension rather than changing suddenly. Consider a shared library that contains an entry *open* whose interface needs to change. We can add a new *open* entry to the end of the function dispatch table, and all new compiled code can use the new entry with the old name. By preserving the old entry, old code still works with new libraries, and source code changes to fix argument types or other interface alterations can be postponed until the next time someone needs to re-compile the code. If an implementation re-links the symbol *open* whenever an application loads a shared library, all new and old compiled code will see only the new interface. This means that all the uses of *open* everywhere would have to change at once whenever we alter the interface.

Prototyping

We have implemented two prototype shared library implementations using object concepts; the first used PC-relative PIC and relied on GCC's inline function feature to implement PIC linkage, while the second, current prototype uses base pointer PIC and has built-in symbol look-up functions. One or two more iterations of prototyping will probably be necessary before we develop a version that is good enough to appear in 4.4 BSD. The description below applies to the existing second prototype implementation, although there may be a few comments that refer to the design for the next prototype.

The Platform

The hardware used for the prototyping is a Hewlett-Packard 9000/340 workstation. The CPU is a 16.67 Mhz Motorola 68030. The system software is a version of 4.3 BSD that contains a few features from 4.4 BSD. The most important feature, without which the prototype would be impossible, is a prototype of a new file mapping virtual memory implementation. This new VM system was ported from CMU's Mach operating system to Berkeley UNIX by Mike Hibler at the University of Utah, and it uses a version of the *mmap* style interface which was originally described in the 4.2 BSD System Manual. Several vendors have implemented *mmap,* including most notably Sun Microsystems, but until now a free version was not available. This VM implementation is currently being ported to the most recent 4.4 BSD kernel source and a revised version is expected to be part of the upcoming Berkeley release.

We implemented the first and second prototypes by modifying the GNU C compiler. It seems likely the further prototypes will also involve changes to the batch linker, in which case we will write our own or modify the GNU linker. The current prototype is based on GCC release 1.36.9; it probably would not be difficult to bring it up to the current release (1.37.1, as of this writing) but the differences are not great.

Position Independence

The PIC implementation is the heart of this prototype. The use of base pointer PIC allows the system great flexibility in laying out the address space and is essential in the implementation of interposition. This subsection covers the strategy for PIC and some of the compiler changes that we made to implement it.

Each library uses two base pointers — one for the function dispatch table and public data, and one for code and private data. Applications that load the library use only the public data pointer; a library uses both pointers. Within an application, PIC is fairly simple. References that are internal to the application use ordinary absolute or PC-relative addressing: absolute for data references, and PC-relative for branches on the Motorola 68000 architecture family. PIC references to public library data are simple indirections through the base pointer with a positive constant offset. Public library functions are accessed through a dispatch table; elements in the dispatch table lie at constant negative offsets from the base pointer. The dispatch table contains offsets that the function calling sequence adds to the base pointer to obtain the addresses of function entry points (see the subsection on relocation below for an explanation of why offsets were used). Library code accesses public data and functions in the same way that application code does, but uses the private base pointer to refer to private data. Since GCC and the GNU assembler (GAS) generate PC-relative addressing for local branches (Bcc opcodes) and function calls (BSR opcodes) on the 68000 architecture, branches are already position independent.[2] Function and data offsets are fixed at compile time, so there are no pointers in PIC text that require changes at run-time and thus there is no need for a table of indirections to implement pure text. References to data from PIC therefore require just one indirection instead of two, which cuts cost and code size. Since references to data go through an arbitrary base pointer rather than the PC, the run-time system can put the data at an arbitrary offset from the code that references it; this becomes important when interposition is implemented (see below).

The compiler changes are reasonably straightforward. The architecture of GCC divides neatly into a language-dependent front end that reduces source code to decorated 'tree' data structures, and a language-independent back end that operates on an 'RTL' intermediate code and generates assembly. We changed the interface between the front end and the back end to detect declarations, initializations and uses of shared library symbols and emit alternate RTL code. The first step was to create a PIC data structure; this structure includes pointers to the name of the shared library, a symbol lookup function for that library, information about the base pointer and a PIC data structure for the base library (used to generate references to private data in libraries). We changed the standard parse tree structure for declarations so that it includes pointers to a PIC data structure and to RTL for an offset.

The next step was to add routines to identify C declarations for functions and data that require base pointer references. When a user requests base pointer PIC for a specific library with a command-line option to GCC, the compiler allocates a PIC data structure for that library. At the point where the parser sees a declaration for a particular function or data element, it uses the PIC symbol lookup function for each requested library to find out whether the declared name represents a function or data element in a shared library. If it is a shared library name, the PIC routines compute the base pointer offset and amend the declaration to record the PIC data structure for the library and the offset. If the

[2] There is one exception — unless GCC sees the flag *-fno-function-cse,* it will sometimes load the address of a function into a register and call the function indirectly. Since it is difficult to arrange for the load to be PC-relative, we currently require this flag when compiling PIC.

user supplies a command-line option to GCC indicating that a specific shared library is being compiled, then private data elements as well as public functions and data elements are also set up for base pointer PIC. Since applications and libraries don't need to agree on the order of private data elements, the compiler generates a symbolic offset to such elements and lets the assembler and batch linker choose the eventual fixed offset.

Given this information, generating base pointer linkage is not too difficult. There were four cases in GCC's expression handling code that we needed to alter: one for ordinary references to code and data, one for calls to code, and two for references to aggregate constants. Aggregate constants get special attention because they may need base pointer PIC linkage from within a library, but they are never declared and thus they are never processed by the PIC functions for declarations. The four cases boil down to two similar PIC routines. The routines first look at the PIC data structure for the appropriate library and arrange to load a base register if this hasn't already been done in this function; the RTL for the base register is saved so that it can be reused later. A consequence of this strategy is that functions that don't refer to a shared library don't suffer the expense of loading a base register and can use a full set of registers; this represents a trade-off against dedicating a set of base registers for particular libraries. The base pointers are loaded using explicit PC-relative addressing; unfortunately the compiler still needs to use a modified version of GAS that we produced for the first prototype in order to process these references, since they are fairly complex. After locating a base register, the routines build RTL for the reference, combining the base register with the offset and adding a dereferencing operator. If the reference is to a public library function, the routine interpolates an extra step that extracts an offset and adds the base pointer, resulting in two indirections.

The code that the compiler generates for shared library references is actually quite compact. An application like *ls* might contain this line of C code:

```
fprintf(stderr, "ls: out of memory\n");
```

The actual assembly code that the modified GCC would produce looks like this:

```
LC10:
      .ascii "ls: out of memory\12\0"
       ...
      movel pc@(___LIB__libc-.:l),a4
       ...
      pea LC10
      pea a4@(96)
      movel a4@(-436),d0
      jbsr a4@(d0:l)
```

The first instruction uses PC-relative addressing to load the base pointer into a base register, `a4`. Subsequently the program makes a call to the C library function `fprintf` with the arguments `stderr` and the string "ls: out of memory\n". The file structure `stderr` lies at offset 96 from the public data pointer for the C library, so the compiler pushes the effective address `a4@(96)` onto the stack. To make the function call, the program goes to dispatch table offset –436, effective address `a4@(-436)`, and loads an offset into an index register, `d0`; using an indexed addressing mode, `a4@(d0:l)`, it then adds the base pointer and the index to get the address of the `fprintf` entry point and transfers to that function.

Relocation

Run-time relocation is as important to position independence as PIC linkage. In this subsection we will run through the relocation types and the rationale for the style of relocation we chose, then examine the compiler changes that we made to implement relocation and initialization of public data (since public data turns out to be a related topic).

The first prototype implementation simply used the standard relocation tables generated by the assembler and linker. Unfortunately the second prototype distinguishes relocation types that are peculiar to its form of PIC, so the compiler was modified to emit these tables as assembly code. The three relocation types in the current prototype represent references to private data, public data and public functions. Private and public data relocations differ by whether the private base pointer or the public base pointer is added to an unrelocated pointer at a given address to generate the correct value of the pointer. Since functions are handled by looking up offsets in a dispatch table, their relocation requires the summing of a base pointer and a dispatch table entry. Since all of the relocations are 32 bits on the 68000 architecture, the actual code to perform relocation can be very tight. Entries for each relocation type are organized into separate tables, and customized assembly code manipulates the table entries and pointer values. The inner loops for data relocations are only three instructions; for function relocations, six instructions. Currently we do not support some useful relocation types including public pointers to private data, private pointers to public data and (a very thorny issue) library pointers to application data. We expect to revamp the relocation strategy for the next prototype, keeping the idea of processing each relocation type in batches but increasing the number of supported combinations.

There is one other relocation type that deserves special mention because it may not be obvious that relocation is occurring. The function dispatch table consists of offsets to function entry points rather than complete pointers. By adding the base pointer to these offsets at call time, the code is actually performing relocation. One reason for doing this is that in a typical shared library there are many functions that are never called during the execution of a typical application; if all the function entry points were relocated at start-up time, the application might end up paying for many functions that it never used. For example, in the shared C library there are 553 entry points that would cost at least one add per entry to relocate — this implies that a relocated dispatch table only breaks even after 553 library calls. Moreover, relocating the dispatch table would force a COW fault on a 4 KB page, and worse yet it renders the dispatch table unsharable. For smaller libraries (or for C++ objects), the break-even point is lower and it probably makes sense to use relocated dispatch tables; this could be set on a library-by-library basis. For large libraries, however, it may be useful to identify other relocations that could be handled by postponing relocation until the time the datum is used. The X library from MIT Project Athena contains huge tables of initialized character pointers that translate error codes into English. Since most X programs use these error strings very infrequently, it makes sense to avoid relocating them in the usual way.

The compiler changes to support initialization and relocation are not very pretty and need some redesign. Most of the ugliness in the GCC support stemmed from a desire to avoid changing the assembler and linker to support new features. Basically what we ended up doing was to generate relocations and initialized public data as assembly code into separate output files. We added two functions to create files to contain relocations and initializations and two functions to walk parse trees and RTL looking for initialized pointers and generating relocations. We modified the function that emits a definition for a variable so that after it decides that it must actually output some assembly code for the definition, it checks in with a PIC function. The PIC function arranges to temporarily suppress base pointer PIC linkage (preventing code generation in the middle of initializations) and tests the declaration to see if it defines public data in a shared library. If so, the function opens a file to contain the initialization and switches this file with the standard assembly output file. It emits a `.org` directive to fix the address of the initialization at the offset previously established for that data element, and it sets a global variable to warn about the switch. After emitting an initialization for library public data, the compiler calls another function to close the initialization file and return to the normal assembly output file, and to re-enable PIC linkage. The GCC function to assemble variables in turn calls a function to output constant data; we altered this function to call a PIC function to walk parse trees looking for initialized pointers and generating relocations into a relocation output file. The compiler emits a label into the assembly code immediately before the pointer initialization, then appends the value of the label to the relocation file. For function pointers, the compiler also appends the dispatch table offset. The function 'globalizes' the label to make it visible to the batch linker when it links the relocation table with the compiled file. Pointer initializations in public data provide a little twist — the problem is that a pointer that belongs in public data may point at another data element from the same source file that has 'static' storage class. Since the pointer initialization is emitted into a different assembly file than the target of the pointer, the compiler can't guarantee that the target of the pointer will be in scope when the library modules are linked together. To get around this difficulty, the compiler generates a global name in the main assembly file and uses a `.set` directive to equate this global name with the target of the pointer; the initialization uses the global name and there are no linking problems.

Bootstrapping

Currently the start-up code for an application that uses shared libraries is very simple. This is more a reflection of its neglect than its real complexities; I will describe what the current prototype does and outline our expectations for the next prototype.

Applications still load the binary file *crt0.o* at the bottom of their address space. This file ordinarily performs the initializations needed to run a C program; our version of it adds just enough to bootstrap the shared C library. The code opens the library file, loads it into the address space with the *mmap* system call, then begins poking through the header. It computes the size of the text segment of the library and uses *mprotect* to write-protect it. It reads a header extension word to recover the offset to the library's public data and initializes the application's copies of the private and public base pointers, then calls the C library constructor function with the base pointers, the environment list and the address of the end of the application's data segment. After successfully mapping the library, its file descriptor can be closed. To run the program, the start-up code calls the standard C *main* function and passes its return value to the C library *exit* function. If any of the system calls fail when setting up the library, the start-up code simply aborts with a core dump. Three small routines are linked in from the regular C library, *open, mmap,* and *mprotect;* we could leave out *mprotect* with a little work. This is really the bare minimum of support needed to start an application up with the C library; the shared library start-up code uses only four system calls in addition to the usual *exit* call that appears in the standard start-up code for statically linked programs.

The C library constructor is also very crude. After initializing the library's own copies of the base pointers, it relocates private data, public data and public function references, respectively. The constructor fetches the relocation tables using symbolic offsets from the private base pointer; these offsets are resolved during the batch linking phase. The constructor uses the address of the end of the application's data segment to initialize state for the *sbrk* interface to the *brk* memory extension system call; we would prefer that *sbrk* and the *malloc* memory allocator get new pages from the end of the shared library data area, but this suffices for now. The constructor uses the environment pointer to initialize the global *environ* variable.

This interface is so primitive that it doesn't allow for linking to shared libraries other than the C library, so some changes are obviously going to appear in the next prototype. The most significant of these changes is a dynamic loader function; this function will be responsible for reading a list of library dependencies for the application and installing and initializing each library. Libraries that require other libraries will recursively call the dynamic loader from their constructors. The loader will try to manage dependencies so that different requests for the same library link to the same image, unless explicitly requested otherwise. We will probably embed the loader in the C library as an efficiency measure, so that most ordinary programs will only need to map one file. We will have to make changes to batch linking to build the dependency list. Currently we do not support run-time relocations in the application; a separately callable relocation function will allow the start-up code to take care of any initialized pointers in the application that refer to library symbols.

Interposition

Presently the only library that may be interposed is the C library, but there are many interesting things that we can do with just that library. I will cover the current scheme for interposition and illustrate it with an example (incomplete as of this writing, alas). I will also give a few details from the design for the next prototype that will fix some of the deficiencies.

The bootstrap code maps an interposing library in the same way as it would map any other library. The differences start when the bootstrap calls the interposing library's constructor. The interposing constructor maps the base library and calls its constructor, providing the private base pointer for the base library and the public base pointer for the derived library. When the base library's constructor returns, the derived library maps or copies the base library's relocated public data, makes any necessary changes of its own to the data, then performs its private initializations. When the derived library's constructor returns, the application runs as usual. The next prototype may differentiate between the pointer to public functions and the pointer to public data so that derived libraries which do not make substantial changes to base library public data don't have to copy the data.

The derived library's function dispatch table differs somewhat from a base library's table. Functions that the derived library re-defines are implemented in the usual way, using dispatch table entries that contain offsets to the text segment of the derived library. Table entries for functions that are not re-defined are initialized with offsets to code stubs in the derived library; when control lands in the stub, the code stuffs the offset to the table entry into a register and jumps to a 'delegation' function. The delegation function uses this offset and the base library's private base pointer to get the proper dispatch table entry from the base library, then computes the address of the actual function, subtracts the derived library's public base pointer from the result and patches the derived library's dispatch table with the offset to the base library function. This run-time linking makes the derived library independent of the location of the base library code.

An example of a derived library is the C library tracing package. In this package, the code stubs jump to a trace function instead of directly to the delegation function. The trace function tests a bitmap to find out whether the library function has tracing enabled; if not, it jumps to the delegation function. If tracing is enabled, it jumps to a formatter. The formatter parses a string of formatting directives (reminiscent of *adb*) and prints argument values into a trace buffer on the stack, flushing the buffer to a trace file when it is complete. It then copies the arguments down on the stack and calls the base library function; on return, it parses more formatting directives and prints any returned values (possibly including values that were altered using pointer arguments). At return time, the proper return value (if any) is placed in the return register and the formatter returns to the application. (This before-and-after processing of library functions is very vaguely reminiscent of the 'method combination' available in CLOS [Bobrow88].) The constructor to the tracing library is responsible for looking in the environment list for tracing configuration options, setting the bitmap to enable or disable tracing on the several hundred C library functions, and opening the trace file. Most of the tracing package's workspace is allocated on the stack so that it can be re-entrant; hence there is no global tracing buffer, for example.

Building Libraries and Applications

Here is where most of the unfriendliness[3] in the second prototype can be found. Since library and application construction is perhaps the weakest aspect of this implementation strategy compared to classic shared library implementations, it will be

[3] NB: The term *brain-damage* was replaced in copy-editing.

necessary to clean this up greatly before inflicting a working prototype on users. Most of the difficulties arose from a desire to postpone work on the assembler and linker. These two programs will probably need less work than they would for a fancy dynamic linking scheme, but the amount of work is still substantial; moreover, committing to a new linking scheme and potentially a new object format will require a consensus with the implementors of 4.4 BSD and perhaps the Free Software Foundation and other interested parties.

Building applications is the easy part. A GCC flag *–fbase-pic-*library selects a library to link to; for example *–fbase-pic-libc* links to the C library. More than one pre-linking flag can be given; the compiler uses the order of the flags as a compile-time interposition ordering. Currently the flag is set by selecting a version of 'cc' to compile with; it should be trivial to change this to use an environment variable. When a small application is linked, it is possible to make it even smaller by using the **–n** flag of the UNIX linker to force the system to load the program all at once instead of on demand from the binary file. This technique makes it unnecessary to align code on page boundaries in the binary file — with 4 KB pages, this often saves a substantial amount of disk space. Two features for building applications are missing from the batch linker; they are not used by most programs but are essential to a complete implementation. The first is run-time relocation for applications. Some application program may need to use the address of a library function or data element in a static initialization; the linker must automatically find the relocation tables for these initializations and arrange for relocation to be performed at bootstrap time. The shared C library will provide a generic run-time relocation function that the bootstrapper (and libraries) can use. Almost all of the references to shared library addresses that have so far appeared while compiling ordinary UNIX applications were in code rather than data, and the compiler can handle this by generating code to sum a base pointer and an offset dynamically. The other missing feature is automatic interposition of library functions. There are a few UNIX programs that redefine some C library functions. The proper behavior of the linker should be to automatically generate a 'derived' library and install offsets to the application's version of the function. So far, we have dealt with these programs (*adb, csh* and so on) by using macro substitution to change the name of the re-defined function; for example, *printf* was changed to *xprintf.* This defeats interposition but this has not proven to be a problem in practice.

Building libraries is not so easy. There are some neat hacks in the current prototype but they don't really cover up the fact that a new linker and a new binary file format are needed. One of the most obnoxious problems is that assembly source files for a library are typically not written to be position-independent. We wrote a set of macros to implement position independent references in assembly and changed the C library sources to use them. It might be possible to add a PIC flag to the assembler that would handle this automatically the way the C compiler handles it, but I suspect that it is difficult or impossible with the current compiling system. The next most obnoxious problem is sorting through the library's external symbols to find the names that really are exported, and choosing an order for them. Many symbols in the C library are used to communicate between compilation units but are not meant to be seen by applications. Once the list of public names has been chosen and their offsets have been assigned, the library builder must generate a perfect hash function for library names using the GNU perfect hash generator. With a big library like the UNIX C library, it can be fairly tedious to identify just the right set of inputs to the perfect hash generator that will allow it to discriminate the many redundant names. The C library build was made more difficult by the fact that our system's C library includes Sun RPC and Yellow Pages functions. Once the perfect hash function has been generated, it needs to be included in the PIC code for the compiler so that it is available during shared library compiles. One idea we have for the next prototype is to make the compiler map these symbol look-up functions as shared libraries, so that they don't have to be included into the PIC sources.

Compiling a library is not too different from compiling an application. A flag *–fbase-pic-default-*library tells GCC that it is compiling a particular library and it must generate code for private data references that is relative to that library's private base pointer; for example, *–fbase-pic-default-libc* tells the compiler that the given source file is part of the C library. An environment variable BASEPUBDIR tells GCC the name of a directory in which to put the assembly files containing pieces of relocation tables and chunks of initialized public data. Given these two changes, *make* can build a library archive in the usual way; when the build is complete, a separate procedure is responsible for converting the archive into a sharable image. Since we decided not to change the linker, we use an *awk* script, a shell script and *cat* to synthesize the function dispatch table, the relocation table and the public data segment. The *awk* script builds the function dispatch table by extracting a list of functions and offsets from the perfect hash generator's input file and printing pointers to the functions in the right order into an assembly file. These function pointers turn into offsets during the batch linking phase (see below). The *awk* script is careful to pad the table on the bottom so that the top is page-aligned; this ensures that the table is sharable. The shell script runs through each relocation type, concatenating the appropriate relocation files that were produced by compiling the various sources in the library and appending zeroes as delimiters. We provide some assembly glue so that the library constructor can get a handle on the relocation tables. We use a trick to get *cat* to build the public data segment — the names of the assembly files that contain public data initializations are encoded so that they lexically sort by their offsets from the public base pointer. This lets us use the shell to glob the files in the proper order. Once these three files are assembled, along with the library constructor source, we link against the archive. The dependencies between the functions and the public data, together with internal dependencies between compilation units in the library, ensure that all the essential code and data are gathered into the image. Another trick enters into play at this stage. We use the UNIX linker's –T flag to set the origin of the image (zero address) to the top of the function dispatch table, which is also the bottom of the public data segment. This causes the linker to lay out the image with the text at the bottom at negative addresses, followed by the dispatch table, then (at positive addresses) the public data and the private data, respectively. Since the library's base pointer points at the bottom of public data, this conveniently turns all the relocatable addresses into offsets from the base pointer. This is especially useful for references to private data, since these use symbolic offsets — the compiler has already generated code to add the private base pointer to the symbolic offset, and now that the linker has shifted the addresses of private data elements to make them relative to the base, the values of the offsets are correct. Similarly, the linker has magically converted all the function pointers in the dispatch table into (negative) offsets from the base. After linking the image, we remove the symbol table and extend the image with zeroes to create a blank common segment. Finally, we patch the header with *adb* to add an explicit offset to the base from the start of the file. We could arrange for the bootstrap code to compute the offset instead, but this was easier, and we soothed our consciences by promising to change the binary file format for the next prototype.

Performance

Performance is not so much a goal of the implementation as simplicity and extensibility, but nevertheless the statistics for space and time savings are very good. **Table 1** presents a list of some small and some large UNIX programs and gives figures to show how much shrinkage occurred by sharing the C library. Absolute sizes of some small programs decrease to shell script proportions.

A great deal of the savings for very small programs is due to using the **–n** flag with the UNIX linker. The very smallest programs could be made to fit inside a 512-byte filesystem fragment by moving more of the bootstrap code functionality into a shared library.

Table 1

	size in bytes		
program	shared	normal	%
tty	516	16384	3.1
yes	532	16384	3.2
pwd	568	16384	3.5
who	1156	24576	4.7
cat	2212	20480	10.8
ls	5864	28672	20.5
egrep	6924	20480	33.8
login	7088	53248	13.3
compress	7136	24576	29.0
finger	10108	49152	20.6
find	10904	36864	29.6
sed	12804	28672	44.7
make	16696	36864	45.3
sh	20720	28672	72.3
deroff	21212	40960	51.8
adb	40076	57344	69.9

Run-time start-up overhead is quite small, too. The bootstrap code executes 4 system calls — *open, mmap, mprotect* and *close* — and when mapping the shared C library it executes 993 instructions, compared to 23 instructions for an unshared binary.[4] This is relatively cheap compared to fancy dynamic linking systems. Many simple, short-lived commands under SunOS 4.0.3 take much longer to run with dynamic linking; for example, we counted instructions when running the *ls* program on a Sun-3 and found that it ran over four times as long when compiled for dynamic linking as when compiled for static linking. For that matter the bootstrap code for the null program (just calls *_exit*) used more than 2.5 times as many instructions as the sample static *ls*. When listing the same directory with shared and unshared versions of *ls* on our HP machines, the ratio came to about 1.06:1.

An argument can be made that dynamic linking time is basically irrelevant in a modern computing environment. If a user spends all of their time running heavyweight processes such as X servers and clients, Emacs, Lisp and so on, the overhead of linking is insignificant compared to the overhead of the program. On the other hand, traditional interactive computing isn't dead yet. A survey of the accounting on our undergraduate machines shows that these have a job mix that includes many short-lived

[4] Instruction counts are a cheap way of measuring short stretches of code. They reflect user CPU time only indirectly, but given a generic instruction mix, counts are not an unreasonable measure of time. To generate the instruction count figures in the text, we wrote a program that single-steps programs using the *ptrace* system call and counts the number of steps.

programs like *ls*. If small, short-lived programs run a few times slower with shared libraries than they did before, there is a measurable impact on a system like ours.

A better reason for wanting a fast, simple implementation is that it could make sharing so cheap that it becomes unnecessary to consider the cost. We think the computing environment could change radically if mapping an object from a disk file became a standard way of collecting tools into a user's workspace. As an example, consider typed data. The data in a stream could contain embedded in it the name of a type that handles that data. A process could handle data of various types without any code of its own for processing the data, by extracting the type and mapping in an object that manages the data. Since virtually every process could map an object in order to read a file, fast loading is absolutely essential to making the technique a success.

Future Work

We plan to keep extending our shared libraries implementation into new areas. The next prototype will include a dynamic loader function that manages the address space of an ordinary UNIX process that needs shared libraries. Library versioning will be implemented through the dynamic loader, which will have the capability of scanning for filenames. The next prototype will also contain a new binary file format that supports arbitrary sections and segments, which we will use to increase locality of reference. For example, we will arrange to generate data elements containing initialized pointers into their own section of the library data segment, so that we can limit COW faults during relocation. We can use the same feature to coalesce constants and combine relocation tables. In the compiler, we will generalize the current PIC scheme to handle more-or-less arbitrary forms of nonstandard linkage instead of just base pointer PIC. Code to handle linking can be mapped into the compiler at run-time like a shared library, so that the compiler itself need not contain cruft for specific shared libraries. This feature will also prove useful for interacting with compiled code from other languages, since many compilers for sophisticated object oriented languages use complex function call or data reference conventions that differ from the traditional calling convention for a particular architecture. An extension of this technique could also be used to 'pre-compile' include files and avoid re-processing great amounts of standard material on every compile. We plan to make a port to the HP PA RISC architecture. Base pointer PIC is well suited to most RISC architectures, since base register data is usually easier to access than absolute data. We plan to port the implementation to Mach, in order to take advantage of Mach features for multithreading and to help produce new Mach servers. We will implement a number of shared library applications; some of the first that have come to mind are 'remote' libraries where the user can interpose RPC stubs and execute a library on another processor, and lightweight processes, where services in a multithreaded server can come from small 'libraries' mapped in on the fly. We will certainly work on implementing a design for C++ shared objects using the GNU C++ compiler as a base; since many of the inspirations for our work came from C++, it would be wonderful if we could return the favor.

We have some long term ideas for research projects which would start with our PIC implementation. These projects would almost certainly use C++ rather than C, so we would begin by creating a rich environment for C++ development and build new projects on top of this environment. We are interested in persistent objects, for example. A user of a CAGD application might develop a new feature or a new model by incremental extension; the code and data would reside in a persistent object that would evolve as users worked on it. Incremental compiling of PIC would allow users to write and test code on the fly, incorporating it in the persistent objects. A shared type manager object would allow different users in the same group to coordinate development of new types and objects of those types. Sharing would allow C++ code to coexist with other systems such as Lisp interpreters; the resources used by C++ would be managed separately from those used by Lisp and the two would be able to call on each other; *e.g.*, an expert system written in Lisp could call on a graphics toolkit written in C++. Another interesting direction would be cross-module optimization. Base pointer PIC code would benefit greatly from global register optimization. On a RISC architecture, it would also be possible to do global base register segment allocation such that data could be assigned to base register segments in such a way as to maximize the number of accesses through common base registers.

Conclusion

We have shown that by implementing shared libraries as objects, it is possible to produce a system that is simple and fast and yet contains many of the advantages of slower and more complex implementations. In particular, both position-independent code and interposition can be used to profit without requiring the symbol resolution at run-time that traditional dynamic linking schemes perform. Much work remains to be done to bring this work to its full potential, but it is already producing results and stimulating interest.

References

Arnold, J. Q. Shared Libraries on UNIX System V. *Usenix Conference Proceedings.* Usenix Association, Summer 1986, pp 395-404.

Bobrow, D. G., DiMichiel, L. G., Gabriel, R. P., Keene, S. E., Kiczales, G., and Moon, D. A. *Common Lisp Object System Specification.* X3J13 Document 88-002R, ACM SIGPLAN Notices 23, September 1988.

Gingell, R. A. Shared Libraries. *Unix Review* Vol. 7, No. 8 (August 1989), pp 56-66.

Gingell, R. A., Lee, M., Dang, X. T., and Weeks, M. S. Shared Libraries in SunOS. *Usenix Conference Proceedings*, Usenix Association, Summer 1987, pp 131-145.

Leffler, S. J., McKusick, M. K, Karels, M. J., and Quarterman, J. S. *The Design and Implementation of the 4.3 BSD UNIX Operating System.* Addison Wesley, Reading MA, 1989.

Stallman, R. M. *Using and Porting GNU CC.* Free Software Foundation, Cambridge MA, 1990.

Stroustrup, B. *The C++ Programming Language.* Addison Wesley, Reading MA, 1986.

Stroustrup, B. What Is 'Object-Oriented Programming'? *Usenix C++ Conference Proceedings*, Usenix Association, 1987, pp 159-180.

Donn Seeley is a senior systems programmer for the Department of Computer Science at the University of Utah. For a number of years he has maintained the compiler and debugger suite for Berkeley UNIX. His first publication for Usenix several years ago was about some neat hacks he made to the C compiler. He may be in a rut.

USENIX Summer Conference
June 11-15, 1990
Anaheim, California

David F. Bacon, Andy Lowry – IBM T. J. Watson Research Center

A Portable Run-Time System for the Hermes Distributed Programming Language

ABSTRACT

We present our implementation of a portable run-time system for Hermes, a very high level language containing integrated constructs similar to those provided at the system and library level in systems like Mach and SQL.

The Hermes features we will focus upon in this paper are lightweight concurrent processes, the use of ports for typed communication between processes, a typed object store and its implementation via the Unix filesystem, and relational data structures for high-level associative queries.

Introduction

Hermes is a language developed at the IBM T. J. Watson Research Center to support programming of complex distributed systems at a very high level.[Str89a] A prototype implementation of Hermes was recently completed, including a compiler and an associated run-time system, and is being distributed free of charge in source code form.

The Hermes prototype has been implemented on Unix, and was designed to allow easy porting to other systems. One of the challenges we faced in developing the prototype was that of hiding details of the underlying platform from the Hermes programmer. In particular, a number of the features provided by Unix and required by Hermes programs, such as multiple processes, inter-process communication, and persistent data storage do not map easily and securely to the abstract model presented in Hermes. In what follows we discuss these features in some detail, and describe how our implementation bridges the gap between the Unix and Hermes models.

Language Design Goals

Hermes is a general-purpose programming language designed from the ground up for large distributed software systems such as operating systems, hospital information systems, window managers, and network management systems. Such programs share the following characteristics:

large	many cooperating implementers.
long-lived	maintainers may not have access to the implementers.
dynamic	parts of the system will change while it is running.
multi-user	multiple concurrent activities.
distributed	multiple concurrent sites of activity.

In our view, a language for this environment must be modular, reconfigurable, and secure. A very high level of abstraction is also required because it enhances portability, readability, and allows more extensive optimization.

This approach requires a more sophisticated compiler. In Hermes, a single construct often has multiple possible implementations, in which case the compiler tries to choose the best one or the programmer annotates the program with a pragma directive. Pragmas may change the performance characteristics of a program, but not its semantics. This allows concerns of correctness and performance to be addressed separately.

There are two major differences between Hermes and other languages aimed at the same type of problems (such as Ada): the first is that Hermes does not have any explicit pointers or aliasing. Instead, relational tables, records, and variants are used to create complex data structures, and copy-on-write techniques are used to avoid aliasing without large performance penalties.

The second difference is that there is no notion of a "main program" which begins execution, computes some result, and then terminates (relying on the operating system to reclaim its resources). The Hermes model is similar to that of an operating system: processes constantly enter and leave the system, and the "program" is simply the sum total of all of the active processes.

Operating systems generally provide protection and resource reclamation for address spaces, which commonly correspond to processes. In Hermes processes also provide protection and resource reclamation, but without requiring a separate address space for each process. A mechanism called *typestate checking* does most of this work at compile-time.

In order to reach the widest possible audience, Hermes was implemented as a compiler to a portable abstract machine. A previous version of the language, called NIL or Network Implementation Language[Str85a, Bal89a] shares most of the basic features of Hermes and was implemented as a native-code compiler on the IBM 370. Where appropriate we will mention techniques used in this implementation to create a high-performance compiled Hermes-like language.

The Process Model

Major Features

In the Hermes model, a process contains local variables and ports that provide connections to other processes. There are no shared variables, no aliasing, and no explicit concurrency within a process.[1] Process creation and termination occur with great frequency, since in Hermes a process is the basic unit of program modularity, serving many of the same purposes as do procedures in other languages. Furthermore, the program text of a process is specified dynamically at process creation time, so new code can be introduced into a running system with ease.

Processes communicate by sending messages over ports. An inport is a message queue, and an outport is a capability to send messages to a particular inport. Ports are discussed in detail in section 3.

A newly created Hermes process has an *initialization port* (an inport) through which it may communicate with its creator (an output port connected to the initialization port is returned to the creator as the value of the create operation). The initialization port may be used to pass parameters and results, and to exchange whatever additional ports may be required.

Hermes processes are *secure*, meaning that it is impossible for one Hermes process to affect another in a way that the second process could not have anticipated (such as corrupting its storage or causing it to encounter a machine exception). Hermes guarantees security at compile-time through a mechanism called *typestate checking*[Str86a] ; a process executing a typestate-correct program is guaranteed to be secure, and a program that does not pass typestate checking can never execute because it will be rejected by the compiler.

Because security is guaranteed by the compiler, multiple Hermes processes can be executed in a single address space without any danger that they will destroy each others' data. This is a significant advantage over writing applications with library- or system-level thread packages; experience has shown such programs to be extremely difficult to debug, since there is no address space isolation to protect the threads from each other.[Con89a] Hermes thus achieves the security of heavyweight processes with the efficiency of lightweight processes.

Implementation

The Interpreter

The basis of the Hermes prototype is an *interpreter*, written in C and currently running on several variants of Unix (AIX on the IBM RS/6000, 4.3 BSD on the IBM RT, and SunOS on the Sun-3 and Sun-4). The interpreter executes instructions on behalf of running Hermes processes. In addition, the interpreter performs scheduling of multiple logically concurrent processes, and provides centralized memory management.

The "instructions" executed by the interpreter make up an instruction set named *LI* (for *Language of the Interpreter*). The LI was designed to efficiently support all the operations of the Hermes language in a form that frees the compiler from adhering to the structure of the original source.

An interpreter-based design was chosen for a number of reasons, including increased portability, ease of code generation, and the ability to investigate architectural features that might facilitate a Hermes implementation. An interpreter is definitely *not* a prerequisite for language security, as evidenced by the earlier NIL system.

Process Representation

A Hermes process is represented inside the interpreter by means of a *process control block* (*pcb*) containing, among other things:

- a pointer to the compiled program (a table of LI instructions) bound to the running process;
- the index of the next LI instruction to be executed;
- a pointer to the function that interprets instructions on behalf of this process; and
- the process' local storage.

An unusual feature of the pcb is the *interpreter function* pointer. Whenever an instruction is scheduled for execution on behalf of a process, the process' interpreter function is invoked to do the work. For ordinary Hermes processes, the interpreter function is simply one that retrieves the LI instruction indexed by the instruction pointer, decodes its opcode, and then dispatches accordingly. However, "foreign" processes can also be accommodated via this mechanism. In particular, the

[1] It should be stressed that these statements refer to the conceptual Hermes process model; the implementation is free to make transparent use of techniques like data sharing and concurrent execution for performance or other reasons.

current implementation includes a small number of processes written in C that provide access to various Unix services such as file I/O and window facilities.

Such foreign processes look like normal Hermes processes to the interpreter and to other processes (since they may own ports), but their pcbs' interpreter function pointers point to code generated by the C compiler. Foreign processes therefore have direct access to operating system services not available via the LI instruction set. Foreign processes can make use of the other pcb fields, including the instruction pointer and local storage areas, to maintain whatever state is required from one invocation to the next.

Clearly, the foreign process mechanism is insecure and therefore potentially dangerous to other running processes. However, the code implementing these processes is linked directly into the interpreter and cannot by loaded dynamically by user processes. We are thereby able to retain strict control on access to foreign processes. We plan to implement a more general framework for interaction with programs written in nonsecure languages.

Procedure-Like Processes

The Hermes process model admits a wide variety of process behaviors, including processes that provide multiple services and save state between calls. In practice, many processes fall into a category that we call "procedure-like," characterized by the following:

- The process responds to a single type of call.
- No state is saved between calls to the process.

We use the term "procedure-like" because the programs for these processes look much more like the procedures one finds in other languages than do other Hermes processes. The standard sequence is: receive a call, perform some work, return results, and terminate.[2]

One other procedure-like characteristic is made somewhat difficult by the normal Hermes process mechanism, namely ease of repeated calls. If a Hermes program is written in the style just described, the caller must create a new process to run the program before *each* call. Of course, this can be alleviated by wrapping the procedure body in an infinite loop; the process would then loop back each time it returned a call, ready to receive another.[3] This approach would result in somewhat less readable code, because the loop serves only an ancillary role with respect to the function performed by the program.

To make programming of such processes more natural, Hermes provides a special variant of process creation called *procedure creation*. The creator of a procedure is given an output port just as in normal process creation. In this case, though, the port is not connected to the initialization port of a newly created process. Instead, the port is really a trigger for the creation of new processes; each time a message is sent on the port, a new process is created, and the message is queued on that process' initialization port. Thus the caller is relieved of the burden of repeatedly creating the processes it needs to call.

Efficient implementation of procedures is important because the facility is heavily used in practice. To avoid incurring the overhead of process creation each time a procedure is invoked, the Hermes interpreter *pre-allocates* a pcb when the procedure is created. This pre-allocated pcb is called the *primary pcb* for the procedure. When the first call is made on the procedure port, the primary pcb is used to (cheaply) create a process to service the request. At that point the primary pcb is marked *in-use*, and any additional calls made on the procedure port will require the creation of new *secondary* pcb's. Once the process executing with the primary pcb terminates, the primary pcb is no longer in-use and can therefore be reused when the next procedure call arrives. In practice, most procedures are called in a serial fashion, allowing the primary pcb to be used for all invocations.

Process Scheduling

In the current implementation, the interpreter uses a simple round-robin scheduling discipline by maintaining a circular queue of ready processes. In each scheduling cycle the following steps are performed:

1. Execute one LI instruction for the current ready process.
2. If the current process was made to block by the instruction, remove it from the ready queue.
3. If any blocked processes were revived by the instruction, add them to the ready queue.
4. If an asynchronous I/O event occurred, add any processes that were waiting for it to the ready queue.
5. Advance the ready queue to make the next ready process current.

[2] As described in more detail in section 3, the **call** operation provides the primary mechanism for interprocess communication in Hermes, and is the closest counterpart to procedure calls in other languages. The operation causes a message to be sent to another process along with an identification of the calling process. The caller then blocks while the callee dequeues the message, acts on it and perhaps alters it, and eventually returns it to the caller. At that point, caller and callee proceed (in particular, the **return** statement does *not* cause a process to terminate).

[3] The process would eventually terminate with an exception on its **receive** statement when all potential callers had discarded their calling ports.

There are many scheduling policies which may provide superior performance to our round-robin scheduler. Some possibilities include:

- Execute more than one instruction for a process before advancing the ready queue. Much of the scheduling overhead could be amortized over several instructions using this approach, rather than incurring the full overhead on a per-instruction basis.
- Continue executing the current process until it either blocks voluntarily or executes a set number of iterations of a potentially infinite loop. This policy was used in the NIL run-time environment.
- Give an elevated priority to a process executing with the primary pcb of a procedure, thereby increasing the likelihood that the primary pcb would be free when the next call was made to the procedure.

It should be noted that not all context switching requires participation by the scheduler. An example of such a situation is presented below in the discussion of the call/receive mechanism.

Storage Management

The Hermes interpreter manages storage for all processes in a single pool. In the initial implementation, the interpreter used *malloc()* and *free()* each time an object was created or destroyed. When profiling showed this to cause a substantial performance overhead, that portion of the interpreter was reworked to use ''quick-cell'' lists for all allocations of certain sizes determined by profiling. The interpreter now typically uses quick cells for over ninety percent of all allocations. Execution speed improved by a factor of approximately 2.5 as a direct result of this modification.

Each Hermes object is represented internally by a *descriptor-value pair*. The descriptor identifies the primitive type of the object (integer, record, table, output port, etc.) and points to functions that implement the various ubiquitous operations (copy, finalize, equality test, etc.) for objects of that primitive type. A single descriptor is shared among all objects of a given primitive type, so the descriptors reside in static storage.

While descriptive information is not strictly necessary in the object representation (since the Hermes language is strongly typed), its presence allows our implementation to deal with ubiquitous operations like copying and finalization in a simple, generic fashion. The information has also proved invaluable for debugging the interpreter.

The value portion of an object is an instance of a C union type with a component for each primitive type. Some additional descriptive information is included here in cases where that information is not common among all objects of the given primitive type. For example, the value portion of a record object includes the number of components contained in the record, in addition to the data vector containing the component objects themselves.

The semantics of some of the Hermes primitive types allows storage to be shared easily among multiple copies of objects. An example is the Hermes *nominal* type, which provides a mechanism for creating globally unique values for use as tags, keys, etc. The only operations that pertain to nominals are creation, copying, moving, equality testing, and finalization. In particular, once created a nominal may not be altered. As a result, storage for multiple copies of a nominal can be shared without concern for future access conflicts; a reference counter ensures that a nominal persists until all copies have been finalized. Similar treatment is given to output ports and (because most operations on programs are as yet unimplemented) program objects.

A more complicated *copy-on-write* sharing mechanism is employed for table objects. When a copy is made of a table, the storage for the original table is shared and a reference counter is updated. Unlike nominals and output ports, however, a table may be modified. Therefore, each operation that will modify a table must first check whether it is currently being shared (i.e. the reference count is greater than one) and if so, create a new table whose elements are copies of those in the original table (embedded tables will get shared copies as a result). The resulting copy can then be altered without affecting other copies of the original.

The copy-on-write optimization is more difficult for other aggregate types such as records and variants, because operations exist to alter component objects of those types without making reference to the aggregate. Such a modification to an embedded object must be caught and cause the aggregate to be copied first. This problem does not arise with tables because modifications to table elements are disallowed; to update an element it must first be removed from the table, and then reinserted after the update.[4]

Inter-Process Communication

Major Features

Interprocess communication in Hermes is performed via typed objects called *ports*. *Inports* provide logically unbounded queues for incoming messages; *outports* are access rights to place messages on inport queues. Each outport is connected to a single inport, and can be used to queue messages only at that inport. Multiple outports can be connected to a single inport, and messages from different outports are merged fairly into the inport queue.

[4] Notwithstanding this restriction, the compiler is free to generate code that performs updates in place where this is possible, for performance reasons.

Both synchronous (**call-receive-return**) and asynchronous (**send-receive**) communication are supported in Hermes. In synchronous communication, the calling process is suspended until the message is returned (possibly with alterations). The message data is wrapped in a special type of object called a *callmessage*, which identifies the calling process.

Asynchronous communication does not block the sender and does not provide for return of the data by the receiver. For both types of communication, the receiver must actively dequeue the message by performing a **receive** operation on the associated inport. Hermes provides an operation to test whether an inport's queue is empty and the **select** operation to multiplex message receipt from multiple inports by receiving a message from any available queue or blocking until a message becomes available.

All Hermes ports are *typed*, meaning that they are only capable of transmitting data of a particular type and typestate. Not only does this allow the Hermes language to remain secure despite the possibility of interactions between separately compiled processes, it also relieves the programmer from the need to convert data to and from some canonical serial form such as a byte stream. Integers are transmitted and received as integers, and records as records; even large tables can be communicated without extraneous manipulation. Furthermore, ports are first-class objects, and can therefore be communicated among processes over suitably typed ports as easily as integers. In particular, communication of outports provides many of the features of capability-based systems[Lev84a, Wul81a], but because Hermes is a secure language, costly run-time validation checks are unnecessary.

Because Hermes processes cannot share data, their only means of interaction is the passing of messages. One result of this is that Hermes programs can be distributed across multiple systems with ease, requiring only that the run-time system support passing of messages between machines. Furthermore, since the granularity of a Hermes process is roughly that of a procedure in other languages, relatively fine-grained distribution can be achieved with little work required by the compiler.

Distribution decisions are abstracted away from the programmer, who structures his programs without regard to machine boundaries. Instead those decisions are made either by the compiler or the run-time system. In contrast, a system like Mach provides tasks, threads, and procedures as units of program decomposition, and if the programmer decides that what was previously a single-threaded task should now be a multi-threaded program distributed across several machines, significant rewriting will be required.

Example

To illustrate the constructs for inter-process communication, we present a simple "grep" process. The process is initialized as a server containing the string to grep for. It is given a capability to a line-oriented output function, PutLine, and returns a new capability of the same type which is bound to the grep process.

The callmessage interface definition is shown below: the first two parameters, PutLine and String, are input parameters and are declared as constants. The third parameter, ClientPutLine, is the new capability that is returned. The exit declaration says that the callmessage will be "full" on exit, meaning that all parameters will be fully initialized.

```
FilterDefs: using (stdio) definitions
  Filter: callmessage (
      PutLine: PutLineFn,
      String: Charstring,
      ClientPutLine: PutLineFn
  ) constant(PutLine, String)
      exit {full};
  FilterQ: inport of Filter
      {init(PutLine), init(String)};
  FilterFn: outport of FilterQ;
end definitions
```

The inport declaration includes the type of objects that will be received, namely Filter callmessages, and the typestate (the initialization level) of the parameters on entry to the inport. In this case, the first two parameters are initialized, but the final parameter is not.

An example of the use of the grep process is shown below. First the grep process is created, and a capability to its initialization port, of type FilterFn, is returned and assigned to makeGrep. makeGrep is then called, passing the putLine capability from the standard environment and the string "Hermes". The grep process takes these two objects and creates a new port to itself, returning a capability which is assigned to greppedPutLine. Finally, the client process iterates over a table of strings and outputs them via greppedPutLine, so that only the strings containing "Hermes" will be displayed.

```
grepUser: using (filterDefs,stdio)
  linking (grep) process (q: mainQ)
declare
  makeGrep: FilterFn;
  greppedPutLine: PutLineFn;
  string: charstring;
  textFile: charstringList;
  ....
begin
-- some initialization code goes here

-- create "Hermes" grepper
  makeGrep := create of process grep;
  greppedPutLine :=
    makeGrep(stdEnv.putLine, "Hermes");
-- now grep textFile for "Hermes"
  for string in textFile[ ] inspect
    call greppedPutLine(string);
  end for;
  ....
end process
```

The directive "linking (grep)" tells the compiler to statically link in the code of the previously compiled grep process. Processes can also be dynamically loaded, provided that the creator has received the capability to load a previously compiled program, or has created a new program dynamically. In that case, the create statement would be

```
  makeGrep :=
    create of stdenv.load("grep");
```

Finally, the grep process itself: the callmessage is dequeued from the initialization port with **receive**, and copies of the necessary constant parameters are made. Then a new inport is created and the outport args.clientPutLine is bound to it with **connect**. The initialization message is then returned, and the grep process goes into a **while** loop waiting for output requests.

In Hermes, a **return** does not terminate a process; control simply continues with the next statement. Also, the parameters of a call are treated as components of a single callmessage object, instead of each actual parameter having an associated formal parameter. This is because a process can handle any number of calls simultaneously.

```
grep: using (stdio,filterDefs) linking (inString)
  process (q: filterQ)
declare
  args: filter;
  putline: putLineFn;
  string: charstring;
  grepQ: putLineFn;
  grepArgs: putLine;
  inStringProc: stringPredicateFn;
begin
-- initialize putline and string
  receive args from q;
  putline := args.putline;
  string := args.string;
-- create grepQ service port, return capability:
  new grepQ;
  connect args.clientPutLine to grepQ;
  return args;
-- create inString procedure
  inStringProc := procedure of process inString;

-- handle repeated grep requests:
  while 'true' repeat
    receive grepArgs from grepQ;
    if inStringProc(grepArgs.output, string) then
      call putline(grepArgs.output);
```

```
    end if;
    return grepArgs;
  end while;
end process
```

Implementation

The implementation of intra-interpreter communication is straightforward. Inports are represented as linked lists of queued messages, and outports are simply pointers to inports. The only semantic subtlety is that inports must be fair: no client outport may be "starved" because other clients are flooding the inport with messages. But because inports are FIFO queues, the round-robin policy of the scheduler guarantees fairness.

Although Hermes programs always use message passing with call-by-move semantics at the logical level, several different parameter passing mechanisms are used in the implementation. For single-word quantities such as integers and outports, parameters are passed by copying the value into the callmessage on call and copying it back on return. For larger objects such as records and inports, a pointer to the object is passed. Since the compiler insures that an object can only be accessed by one process at a time, parameters can be passed by reference within an address space without sacrificing security.

In the NIL compiler, performance analysis showed that very large amounts of time were being spent enqueing messages for blocked processes. As a result, inports were modified to contain a function pointer to an enqueue routine. When the process owning an inport is blocked waiting for a message to appear on it, the enqueing function simply saves the current registers of the process, loads the parameters into the registers, and jumps directly into the called process. The result is an eight instruction context switch between protected processes.

The implementation of inter-interpreter communication is more complex, although the high-level abstractions of Hermes make this task considerably simpler than in other languages. There were a number of Unix limitations that had to be overcome, most notably the limited number of file descriptors and the lack of asynchronous communications facilities.

Distributed Hermes uses Sun RPC and XDR protocols over TCP for communication. For each Hermes data representation, there is an associated XDR encode/decode routine which is automatically invoked by the run-time system for a remote call, return, send, or receive. Because there are no explicit pointers in Hermes, the only pointers are those created by the run-time system to implement the Hermes abstractions, which makes it easy to locate all the portions of a data structure for transmission.

Because a Hermes interpreter runs many processes concurrently, it is not acceptable for it to block during a remote procedure call, particularly since the callmessage might never be returned! It is also not acceptable to block during a receive from a remote port, since there may be other runnable processes.

To accommodate these requirements, we created RPC procedures layered on top of the existing synchronous Sun RPC calls. They call the regular routines and then set the input file descriptors to generate SIGIO interrupts when data is available, and set the output file descriptors to suppress buffering of packets.

The small number of file descriptors available in Unix also made it necessary to implement an LRU file descriptor cache so that an interpreter could communicate with more than 15 other interpreters. A connection is always used at least once before being swapped to prevent thrashing.

Both the asynchronous RPC and the file-descriptor caching are part of an RPC enhancement library that is completely independent of the rest of the Hermes system.

In the initial implementation, distribution decisions are made on the basis of locality: each interpreter is created with I/O interface processes and some sort of shell for communicating with a user. New processes are instantiated in the same interpreter as the creating process. Remote creation and migration facilities are under discussion. The obstacles are not technical, but conceptual: what is the proper interface to the programmer or compiler for control over such decisions? This question is currently under (heated) debate.

Outports connected to remote inports are represented as triples containing a machine name (internet address), interpreter name (RPC version number), and address of the remote inport in that interpreter. When an outport is sent to the interpreter containing its inport, it is converted back into a local outport.

The Typed Object Store

In most languages, complex data structures created in memory can not be stored on disk without performing some sort of conversion process which marshalls the data structure into a byte stream. Although facilities like XDR and rpcgen[Sun87a] can partially automate these procedures, they generally do not handle pointer-based structures that contain cycles (such as graphs) or other kinds of pointer aliasing, and if some pointers are opaque, user-written routines *must* be introduced.

In the Hermes language, there are no pointers and no aliasing. Although the compiler represents programmer data structures using both pointers and aliasing, this is invisible to the programmer. Since the compiler has full control over the data structures, it was easy to provide run-time support for converting data structures to and from byte streams.

The same representation is used both for disk storage and for network transmission. As a result, Hermes objects (including compiled Hermes programs) can be shared by heterogeneous systems all mounting the same filesystem.

The same XDR routines described in the previous section on inter-process communication are used for converting Hermes objects into files. Details on conversion of table data types are provided in the following section.

The main disadvantage to using XDR for disk storage is the large amount of redundant information stored: applying the Unix compress utility to these files yields a five- to ten-fold compression. We plan to provide an option to store these files in compressed format, allowing I/O performance to be traded off against disk space.

Tables

Major Features

The Hermes *table* is a high level data abstraction for programming with homogeneous collections of objects. The Hermes table is designed to eliminate the need for such traditional user-defined data structures as arrays, pointer-based linked lists and trees, hash tables, and so on. The model is loosely based on the relational data model and allows queries and other operations to be programmed without regard to the underlying physical organization of the data.

Besides creation of tables and insertion of elements into tables, Hermes provides a uniform mechanism for selecting elements of a table to be scanned, removed, or copied out. The *selector* construct is used in each of these operations to identify, via an arbitrary test expression, which table elements are to participate.

Two optional attributes can be included in table type definitions. The first, *ordered*, indicates that the table elements exist in a fixed order relative to one another. The position of each element in this ordering is established (relative to the existing elements) when it is inserted into the table. Concepts like "the first element" make sense only when applied to ordered tables. Ordered tables provide indexed access, such as is normally associated with arrays.

The second attribute, *keyed* implies that a certain portion of each table element forms a primary key (in the database sense) for that table element. The semantic implication is that no two elements in the same table can have the same key value. Multiple keys can be declared for a table type.

Figure 1 shows some code fragments illustrating the high level of programming abstraction afforded by the Hermes table type. Honesty requires us to point out that the final code fragment would perform very badly if compiled with the current Hermes compiler. It is one of the future goals of our project to demonstrate that high-level transformations can be used to produce extremely efficient code from high-level specifications such as this.

Following are the table type definitions used in the sample code fragments of Figure 1:

```
-- a simple ordered table
charstring: ordered table of char {init};

-- an unordered set of distinct nodes
nodeSet: table of nodeID {init}
  keys (*);

-- an edge with an identifying string
edge: record (
  source: nodeID,
  sink: nodeID,
  label: charstring
);

-- an unordered set of distinct edges
-- with unique labels
edgeSet: table of edge {full}
  keys (source,sink) (label);

-- standard specification of a graph
graph: record (
  nodes: nodeSet,
  edges: edgeSet
);

--   a table that is both ordered and keyed
nodeOrdering: ordered table of nodeID
  {init} keys (*);
```

Implementation

Internal Representation

A uniform representation and implementation for all table objects would almost certainly perform quite badly, since no single representation will be efficient for all tables under all circumstances. For this reason, the Hermes run-time system supports a number of specialized implementations. The mapping of table instance to run-time representation is made by the compiler, and could be based on various criteria, including table attributes (ordered, keyed), hints (*pragmas*) supplied by the programmer, and profiling data.

Currently, the Hermes interpreter supports four internal representations for tables:

- Linked list
- Vector
- Character string (packed vector of byte values)
- AVL tree (self-balancing binary tree).

The internal representation of a table object includes, in its value part, one or more *table descriptors* identifying the representation(s) used for the table. A table descriptor contains, among other things, pointers to specialized functions to perform primitive operations on tables in the given representation. Thus, for example, the code to interpret an **extract** statement makes indirect calls to representation-specific *remove()* functions for each element satisfying the extract test. Pointers to the *remove()* functions are found in the table descriptors attached to the table.

The ability to attach multiple table descriptors to a single table means that the compiler can choose to maintain multiple representations simultaneously for a single table. Each representation may be highly efficient for some subset of the operations or queries that are likely to be performed on the table. This is analogous to the capability of maintaining multiple indices for a single relation in a relational database; though such a capability is common in database products, it is quite unusual as an implementation strategy in a general purpose programming language.

Adding a table representation to the implementation consists primarily of the following steps:

1. Design the necessary data structures.
2. Implement each of the required primitive table operations including, as appropriate, operations making use of positions (ordered tables only) or keys (keyed tables only).
3. Make a descriptor for the new representation available to the interpreter.
4. Modify the module of the code generator (which is written in Hermes) that chooses representations for tables, to make use of the new representation where appropriate.

The compiler currently uses a fairly straightforward set of heuristics for choosing table representations. First, a "primary" representation is selected, which is used by any operation that must scan through all the elements of the table:

- If the table is ordered, a vector or charstring representation is used. Charstring is chosen only if the compiler can determine that every potential table element value can be represented in a single byte.
- If the table is unordered and unkeyed, a linked list representation is chosen.
- If the table is unordered but keyed, one of the keyed representations (see below) serves as the primary representation.

After the primary representation has been selected, additional representations are established for every key appearing in the table type definition. AVL trees are currently used for this purpose, though we may soon switch to hash tables.

When a table operation involving a selector is used on a table with multiple representations, the compiler can choose among several access methods to locate the required table elements. Currently, the following strategy is used:

- If the selection test is not a conjunction or a simple test for equality, a loop is generated to scan the table using the primary representation, and each element is tested as it is scanned.
- If the table is ordered and a top-level conjunct in the selector is an equality test involving the position of the element being tested, an indexed vector access is generated. Any remaining tests in the selector are applied to the element retrieved by this probe, if any.[5]
- If the table is keyed, and at least one of the keys is "covered" by the selection test, a keyed lookup using the AVL tree based on that key is generated, and any remaining tests are applied to the retrieved element, if any. A key is covered if each component key field appears in an equality test which is a top-level conjunct in the selection test.
- If neither positional nor keyed access is applicable, the scanning loop strategy is used.

External Representation

A few special tricks are used in converting tables to an external form either for transmission over a net-

```
-- Declarations that would be needed for code fragments
pos: integer;
string: charstring;
junk: charstring;
firstchar: char;

g: graph;
successors: nodeSet;
startNode: nodeID;
unplacedNodes: nodeSet;
sortedNodes: nodeOrdering;
unplacedNode: nodeID;

-- remove initial blanks from string, get copy of first non-blank character
pos := position of char in string where(char <> ' ');
extract junk from char in string where(position of char < pos);
firstchar := string[0];

-- find all immediate successors of the start node
successors := every of node in g.nodes
  where (exists of edge in g.edges
    where (edge.source = g.nodes[startNode] and edge.sink = node));

-- topologically sort a graph
block begin
  unplacedNodes := g.nodes;
  new sortedNodes;
  while (size of unplacedNodes <> 0) repeat
    remove unplacedNode from node in unplacedNodes
      where (not exists of pred in unplacedNodes
        where (exists of edge in g.edges
          where (edge.source = pred and edge.sink = node)));
    insert unplacedNode into sortedNodes;
  end while;
on (NotFound)
--  call putLine("Graph is not acyclic.");
end block;
```

Figure 1: Sample code fragments illustrating programming with tables.

[5] Note that in this case and in the case of keyed access, the indicated method may be disqualified based on the form of the other comparand(s). The details are omitted from this presentation.

work or for disk storage.

First, descriptive information for table elements is stored once for the entire table, rather than once per element. This is analogous to the elision of individual object descriptors in the internal table representation. This strategy is unfavorable only in the case of an empty table.[6]

Second, even if a table has multiple internal representations, each element is encoded only once. Representation descriptors appear at the beginning of the external representation, so that decoding the table can be accomplished by creating a new empty table with the indicated representations, and then inserting each element as it is decoded.

Finally, special treatment is given to tables with a charstring representation. Normally, each element of a table is encoded by invoking the generic XDR routine to encode an object of the table element's primitive type. Because the XDR specification requires a minimum of four bytes per encoded value, this results in three-to-one ratio of wasted to useful space in the case of charstrings. To avoid this, a special check is made for the existence a charstring representation, and if it is present, a single call to a byte-oriented XDR routine is used to encode the entire table at once.

Related Work

Hermes differs from other languages for large, concurrent systems like Ada[DOD83a] and Modula-3[Car88a] primarily in its process model and its substitution of tables for pointer-based structures.

Ada and Modula-3 provide support for concurrency within an address space, but require extralingual constructs to be used for communicating across address-space and machine boundaries. Because of their shared-memory model, transparent distribution is much more difficult.

These languages also do not provide for dynamic compilation and linking of code.

The Hermes model of processes communicating by ports is very similar to that of Mach.[Ace87a] However, since Mach is providing an operating system service, it does not perform type checking across ports and must use address space protection to provide security. By providing these abstractions within the language, we can move much of this work into compile-time typestate checking, allowing multiple programs to execute securely within a single address space.

This frees programmers of the necessity to choose between fast, insecure communication (between threads) and slow, secure communication (between tasks), each with its own programmer interface.

Conclusions

The Hermes language allows distributed applications to be created more easily than existing languages and systems. This is due in large part to its very high level view of communication and computation.

The high-level view also allowed certain parts of the implementation to be much simpler than that of other languages, for instance the object store and the remote communications. On the other hand, many Hermes constructs are very powerful and require substantially more sophistication on the part of the compiler.

The table data type requires the compiler and run-time environment to provide a large portion of database functionality, and to manage the automatic mapping of a single data type into multiple data structures.

Our initial plan was to quickly build an interpreter-based system and then use it to bootstrap a native-code compiler. While we still plan to follow this path, the effort of creating the interpreter was significant enough that in retrospect, our effort might have been better spent on just building a compiler from the start. In addition, the movement toward standardization during the last few years has made many of our original portability concerns, which led us to choose an interpretive design, obsolete.

The Hermes system is designed for exactly the kind of computation environment which is emerging among Unix systems: distributed networks of uni- and multi-processors. The security, dynamics, and powerful high-level operators of Hermes should make it possible to rapidly create applications for this environment. We are making Hermes available in source code form in the hope that others will use it for their own studies of the interaction between computer hardware, networks, operating systems, and languages.

To request Hermes code and documentation, send e-mail to `hermes-request@ibm.com.`

Acknowledgements

The authors wish to acknowledge the many people who have contributed to the Hermes implementation, either through their ideas or with actual code, including Rob Strom, Shaula Yemini, Danny Yellin, Arthur Goldberg, and Joel Auslander.

[6] We have considered, and may adopt in the future, a special representation for empty tables, both internally and externally. A special internal representation would eliminate the overhead of building data structures at table creation time in the fairly common case that the table remains empty throughout its lifetime.

References

Ace87a. Mike Acetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young, "Mach: A New Foundation for UNIX Development," CMU Department of Computer Science Technical Report (1987).

Bal89a. Henri E. Bal, Jennifer G. Steiner, and Andrew S. Tannenbaum, "Programming Languages for Distributed Computing Systems," *ACM Computing Surveys* **21**(3) pp. 261-322 (Sep 1989).

Car88a. Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson, "Modula-3 Report," Olivetti Research Center Report ORC-1 (1988).

Con89a. Daniel C. Conde, Felix S. Hsu, and Ursula Sinkewicz, "ULTRIX Threads," pp. 257-268 in *Proceedings of the 1989 Summer Usenix Conference*, The Usenix Association (1989).

DOD83a. DOD, "Reference Manual for the Ada Programming Language," ANSI/MIL-STD-1815-1983 (Feb 1983).

Lev84a. Henry M. Levy, *Capability-Based Computer Systems,* Digital Press, Bedford, MA (1984).

Str85a. Robert E. Strom and Shaula Alexander Yemini, "The NIL Distributed Programming Language: A Status Report," *ACM SIGPLAN Notices*, (Jun 1985).

Str86a. Robert E. Strom and Shaula Alexander Yemini, "Typestate: A Programming Language Concept for Enhancing Software Reliability," *IEEE Transactions on Software Engineering* **SE-12**(1) pp. 157-171 (Jan 1986).

Str89a. Robert E. Strom, David F. Bacon, Arthur P. Goldberg, Andy Lowry, and Daniel Yellin, "Hermes Language: Tutorial and Reference Manual," To be published by Springer-Verlag in 1990 (1989).

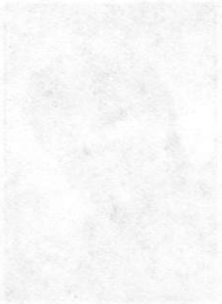
Sun87a. Sun, "Network Programming," Distributed with SunOS 4.0; part number 800-1779-10 (1987).

Wul81a. William A. Wulf, Roy Levin, and Samuel P. Harbison, *HYDRA/C.mmp: An Experimental Computer System,* McGraw-Hill, New York, NY (1981).

David Bacon began programming computers at 12 and got his first programming job at 15. He received his A.B. in Computer Science at Columbia University and now works at the IBM T. J. Watson Research Center, Hawthorne, where he was a designer and lead implementer of the Hermes language. His other interests include opera, fencing, bicycle touring, and transparent optimistic transformations for such problems as fault-tolerance, parallelization, and replication.

Andy Lowry received his B.S. and M.S. and is currently pursuing his Ph.D., all in Computer Science at Columbia University. He is currently conducting his research as a full-time employee at the IBM T. J. Watson Research Center. As a member of the Hermes project, Andy has contributed substantially to the implementation of the Hermes language. He has over ten years experience working as a computer programmer. His research interests include optimistic techniques, especially as related to parallelization, database concurrency control, and replication of data in distributed systems.

David Bacon began programming computers at 12 and got his first programming job at 14. He received his A.B. in Computer Science at Columbia University and now works at the IBM T.J. Watson Research Center, Hawthorne, where he was a designer and lead implementor of the Hermes language. His other interests include opera, fencing, [illegible], and transparent optimistic transformations for such problems as fault-tolerance, parallelization, and replication.



Andy Lowry received his B.S. and M.S. and is currently pursuing his Ph.D., all in Computer Science at Columbia University. He is currently conducting his research as a full-time employee of the IBM T.J. Watson Research Center. As a member of the Hermes project, Andy has contributed substantially to the implementation of the Hermes language. He has over 10 years' experience working as a computer programmer. His research interests include optimistic techniques, especially as related to parallelization, data access, concurrency control, and replication of data in distributed systems.

References

Acc86. Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young, "Mach: A New Kernel Foundation for UNIX Development," CMU Department of Computer Science Technical Report (1986).

Bal89. Henri E. Bal, Jennifer G. Steiner, and Andrew S. Tanenbaum, "Programming Languages for Distributed Computing Systems," ACM Computing Surveys 21(3) pp. 261-322 (September 1989).

Car89. Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson, "Modula-3 Report," Olivetti Research Center Report/DEC SRC (1989).

[illegible] "[illegible] UNIX," pp. [illegible] in Proceedings of the 1989 Summer USENIX Conference, [illegible] (1989).

[illegible] "[illegible]," SIGPLAN [illegible] (19[illegible]).

Lev84. Henry M. Levy, Capability-Based Computer Systems, Digital Press, Bedford, MA (1984).

Str85. Robert E. Strom and Shaula Yemini, "The NIL Distributed Systems Programming Language: A Status Report," ACM SIGPLAN Notices [illegible] (1985).

Str86. Robert E. Strom and Shaula Yemini, "Typestate: A Programming Language Concept for Enhancing Software Reliability," IEEE Transactions on Software Engineering SE-12(1) pp. 157-171 (January 1986).

Str90. Robert Strom, David F. Bacon, Arthur Goldberg, Andy Lowry, Daniel Yellin, and Shaula Yemini, "Hermes Language Tutorial," to be published by Springer-Verlag (1990).

Sun87. [illegible], "Network Programming," [illegible] (1987).

Wul81. William A. Wulf, Roy Levin, and Samuel P. Harbison, HYDRA/C.mmp: An Experimental Computer System, McGraw-Hill, New York, NY (1981).

USENIX Summer Conference
June 11-15, 1990
Anaheim, California

Deceit: A Flexible Distributed File System

Alex Siegel, Kenneth Birman, Keith Marzullo – Cornell University

ABSTRACT

Deceit[1], a distributed file system being developed at Cornell, provides flexible file semantics to control efficiency, scalability, and the desired reliability. The user is able to set parameters on a file to achieve different levels of availability, performance, and file location transparency. File replication, file migration, and a sophisticated update control protocol is supported. Deceit also supports a file version control mechanism. Deceit can behave like a plain Sun Network File System server and can be used by any NFS client without modifying any client software. Deceit servers are interchangeable and collectively provide the illusion of a single, highly available, and large server machine to its clients. Non-volatile replicas of each file are stored on a subset of the file servers. Servers may be grouped into independent subsets, or cells, for security and efficiency. The Deceit prototype uses the ISIS Distributed Programming Environment for all communication and process group management - an approach that reduces system complexity and increases system robustness. Common faults such as lost messages, crashes, and partitions are tolerated.

1. Introduction

File systems are a critical component of modern computing environments. As networks grow, it is becoming important to provide file systems that grow with the network. For example, in an environment with 200 workstations, one file system that runs concurrently in 200 places is more manageable than 200 separate systems. These ‘‘distributed’’ file systems can provide crash resilence and performance superior to traditional file systems.

Most existing distributed file systems are not flexible in the ways files can be managed or accessed. Deceit is a distributed file system, being developed at Cornell University, that provides such flexibility with a wide range of file options. These options allow the user to make decisions about availability[2] and transparency[3]. An advantage of more flexible file options is that performance can be better optimized. A disadvantage is that the file system becomes more complex.

Deceit provides many capabilities to the user: file replication with concurrent reads and writes; a range of update propagation strategies; and a file version system. In addition, Deceit provides Sun NFS protocol compatibility, since NFS has become a *de facto* industry standard. No change in NFS client software is necessary in order to use Deceit. Refer to the appendix for a summary of the NFS protocol.

Many distributed file systems have been developed. A short list includes [5,10,11,16-20,22]. Deceit is unusual for two reasons: great flexibility in the protocols and full NFS compatibility. We are indebted to these other distributed file systems for many of Deceit's design features. A full presentation of other distributed file systems is beyond the scope of this paper, and a partial presentation would inevitably be unfair, so we only compare Deceit to NFS in this paper.

In section 2, we will discuss the environment in which Deceit has been designed to operate. In section 3, we will present the architecture of Deceit. The architecture is divided into two sections: the structure of the program and the main inter-server communication protocols. Then, we will describe of some of the mechanisms that provide fault tolerance in Deceit. In section 4, we will present some performance benchmarks and the results of other analysis. Section 5 concludes the paper.

2. Environment

Deceit is designed to run in the same network environment that NFS assumes. That is, failures such as message loss, crash, partition, and other common types of network and machine failures can occur. A more detailed description of the environmental assumptions follows.

Network Assumptions

The target environment is a network of computers used in a client/server fashion. Some of the computers may be diskless, and some may be large dedicated file servers. Under normal conditions, all

[1] This work was supported by DARPA/NASA-Ames Grant No. NAG-2-593.

[2] File *availability* is the percentage of time that a file can be read or written by the user.

[3] File *transparency* is the degree that file replication and data movement is invisible to the user.

machines can communicate directly with each other through an underlying network. Communication is symmetric: if *a* can send a message to *b* then *b* can send a message to *a.* Network communication is secure: messages are sent to the correct destination with a correct source address, and messages can not be examined by machines which are not the intended receiver. All communication between servers is through the ISIS distributed system[2, 3]. Therefore, all of the ISIS network assumptions are also assumed by Deceit.

Failure Assumptions

We assume that machines may crash without notification; messages may be lost during transmission; and the network may experience long term communication partitions. Network partitions may be frequent.

Operational Assumptions

Predictable file access patterns are central to the design and performance of Deceit. Many of Deceit's design decisions were based on results from studies which were done in an academic environment[7, 8, 21, 23].

Deceit's operational assumptions are as follows. Files tend to be written or read in their entirety with a stream of operations. Nearly simultaneous writes by two clients to the same file are very rare. Files experience long periods of total inactivity punctuated by high activity where they may be rewritten several times in a few minutes. File activity tends to cluster in a small number of directories. The vast majority of NFS operations are *get attribute* (get basic file attributes), *lookup* (find a file by name in a directory), *read,* and *write.*

Related Topics

The ISIS distributed system is used for crash and partition detection[4], communication primitives, and group membership protocols[12]. Some features that ISIS provides are: several group broadcast protocols, atomic group membership change, mechanisms for locating group members by group name, light-weight processes with signals and semaphores, architecture independent communication, and process state transfer. As a detailed discussion of ISIS would be a digression, the reader is referred to [4] for more information.

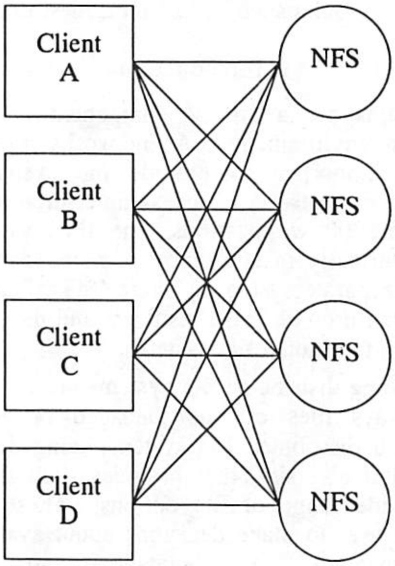
Throughout this paper, the term "server" will be used to refer to a computer with a non-volatile storage system which forms a permanent file repository. The term "user" will be used to refer to the person or process who is initiating file system operations. A "client" is a computer that issues requests to a server and receives requests from a user; clients may also be servers, and one server can act as a client to another server.

3. Main Architecture

The Deceit architecture can be divided into two orthogonal categories: the structure of Deceit as a distributed program, and the communication protocols used within Deceit. Fault-tolerance will be considered in both categories.

3.1 Program Structure

A main difference between Deceit and NFS is that files are not statically bound to any particular server; with Deceit, files may be replicated and move freely around. If a client issues a request for a file to a server that does not have a replica of that file, the request is automatically forwarded to a server that has a replica. The reply to the client is propagated backwards along the same path.



NFS Communication

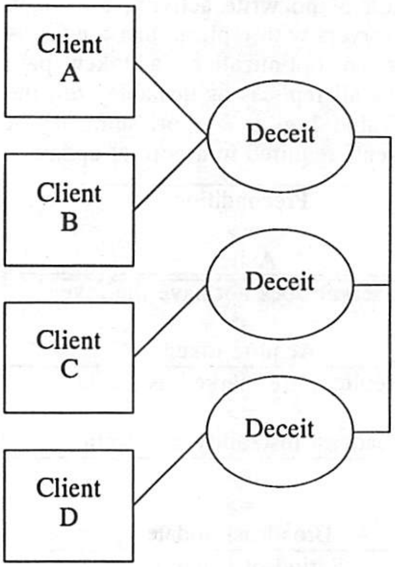
Servers provide an identical file service to all clients so that clients only have to explicitly connect to one server in order to access the entire Deceit service. Furthermore, when one machine fails, Deceit clients can reconnect to another machine and continue operation[5]. Users can think of Deceit as being a single, highly reliable and responsive server. (See Figure 1)

Deceit provides a superset of NFS functionality. To allow the user to access this functionality, Deceit has additional commands and file operations beyond the normal NFS operations. Clients access these features by using special RPCs and by reads and

[4] The current version of ISIS, version 2.0, does not tolerate partition failures. Future versions will support this type of failure since it is surprisingly frequent in large applications. In anticipation of this capability, Deceit has been designed to tolerate partitions.

[5] Standard NFS client software does not allow clients to change servers in mid-stream. We have developed software to overcome this hurdle.

writes to "invisible" control files. These *special commands* include those that list all versions of a file, locate all replicas of a file, modify file parameters, and reconcile directory versions.



Deceit Communication
Figure 1 - NFS/Deceit Comparison

Components

The Deceit server consists of several components as shown in Figure 2. The first component is the *distributed reliable segment service.* The segment service provides a simple, flat distributed file service with no file type information or user specified names. There is no notion of directories or links in the segment service. The segment service implements all of the update, replication, and versioning protocols, and it is the layer where file control parameters operate. Each normal (non-directory) NFS file corresponds to a segment in the segment service. An operation on a normal file is mapped directly into the corresponding operation on a segment. Directories are stored differently.

The second component is the *directory service.* The directory service is responsible for maintaining and propagating information about hard links. To take advantage of the well understood semantics of hard link creation and deletion, specialized communications protocols are used. These protocols are discussed in Section 3.2.4.

On top of the segment service and the directory service is a full NFS file service which is called the *NFS file interface.* The main task of the NFS interface is to translate NFS requests into primitive operations on segments and hard links.

3.2 Communication Protocols

Deceit employs several layers of communication protocol. The underlying broadcast protocols and failure detection are provided by ISIS. Due to the large number of protocols used in Deceit, only the major protocols will be discussed, and they will not be presented in depth.

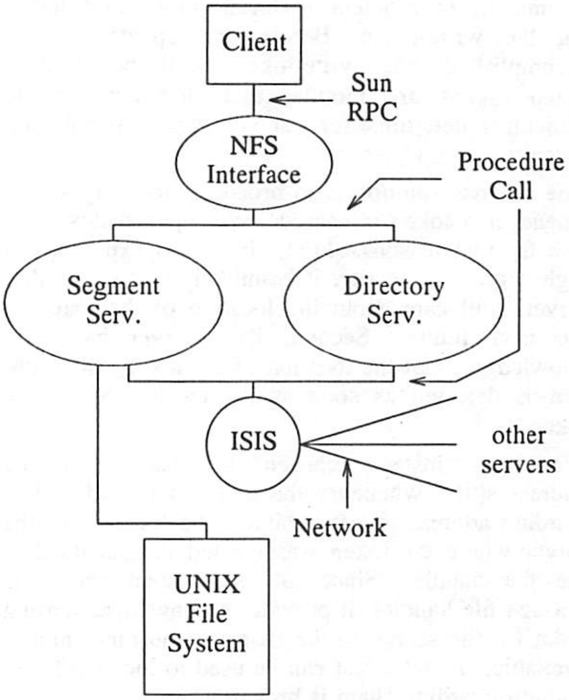


Figure 2 - Deceit Components

3.2.1 Write-token protocol

To coordinate access to replicated files, we use a *write-token protocol.* A *write-token* is associated with each file[14]. Only a server that holds the token is allowed to distribute updates to the corresponding file replicas. A write-token can be considered to be an exclusive write lock on the file, except that a token held by some server at all times. The token holder maintains a list of all servers who have a replica of the file as well as other file state information. This information is replicated at all file replica sites to provide robustness.

An update requires only one communication round[6] if the token is held due to a previous update. A write-token protocol works well when update streams tend to originate from one source for long periods of time as in a file system.

A server that lacks a token must acquire it before distributing an update for that file. Token acquisition requires one round, but it is only done for the

[6] A *communication round* is the distribution of a message to a set of processes. The collection of synchronous replies is included in the round.

first in a series of updates. To acquire a token, a server send a *token request* to the current token holder. The server that holds the token broadcasts a *token pass* to all replicas in response. It is necessary for correctness that the updates arrive in identical order at all servers regardless of token movement.

3.2.2 Finding the Write-Token

A fundamental problem in Deceit is efficiently locating the write-token. Before any update can be accomplished, the write-token must be located. Since tokens are mobile, their location can be difficult to determine for a server which has not been given timely updates.

One obvious solution is to broadcast to every server whenever a token is passed. We rejected this solution for two reasons. First, since files experience a high degree of locality, it is unlikely that most other servers will care about the location of the token in the near future. Second, if a server has stale knowledge about the location of the token, this problem is detected as soon as it tries to acquire the token.

We have instead chosen to use forwarding addresses[9]. Whenever the token is passed, a forwarding address is left behind. The location of the server where the token was created is embedded in the file handle. Since all file system access is through file handles, it provides a convenient starting point for the search in the case that no other hint is available. A broadcast can be used to locate a token if the forwarding chain is broken.

Since the token forwarding chains may become long, this algorithm suffers from *O(n)* time complexity where *n* is the total number of servers. The algorithm can be speeded up by using *path compression.* After a token is located, the new location is sent along the original search path as a location update so that the next search can skip the intermediate sites. It can be shown that this algorithm has *O(log n)* amortized time complexity assuming that no broadcast is necessary.

3.2.3 Stability Notification

Normally a Deceit server responds to a read request by reading the closest available file replica. Since updates can take time to propagate to multiple replicas, without synchronization, a file's data may appear inconsistent for short periods of time. In order to alleviate this problem, Deceit provides *stability notification.*

Before a file is modified, all but one of the available[7] replicas of the file are marked as *unstable.* Unstable replicas are invisible to the user, instead all file reads and inquiries are forwarded to the remaining replica. Only this replica needs to be updated before a write can return to a client to provide apparently instant update propagation for all clients. After a short period of no write activity, the token holder notifies all servers with replicas that the file is *stable* again. As an optimization, a token pass automatically marks all replicas as unstable with the same broadcast. Table 1 gives a short summary of the sequence of events required in a normal update.

#	Precondition => Action
1	The server does not have the token => Acquire token
2	Replicas are marked as stable => Broadcast instability notification
3	true => Broadcast update
4	Period of inactivity => Broadcast stability notification

Table 1 - Steps in an Update

3.2.4 Epidemic Propagation of Directory Updates

In contrast to normal files, Deceit does not store directory contents in segments. There are several reasons for this:

1. Some directories (e.g., "/tmp") are frequently accessed by several concurrent users. If directories were stored in segments, then each server would tend to have a cached copy of these directories. For large numbers of servers, this would lead to very expensive updates.

2. Fault-tolerance becomes complex. Since it may be required to do a full path search to find a file, each file would need to have all of its parent directories on the same server. For example, assume the file *f* is stored at site A. Every ancestor directory of *f* must also be at A, or *f* can become unavailable if every other site fails.

3. Garbage collection becomes difficult[15]. The storage for a file may be deallocated when there are no more hard links to that file. In a centralized file system, it is relatively easy to determine when it is possible to deallocate the storage for a file; for example, a link count can be kept with each file. In a distributed file system, an accurate link count can not be kept with each file because some replicas of a file may be unavailable at any given time.

[7] A replica at server B is *available* to A if A can communicate with B. ISIS provides a clean notion of availability since failure detection is coordinated with communication.

For these reasons, each Deceit server maintains a list of all hard links in a local auto-balancing binary tree. This in-core data structure is backed up on non-volatile storage. Updates are propagated with an epidemic algorithm to all servers[6]. Every 10 seconds, each server picks another random server. Each pair exchanges a list of updates to bring each other up to date. Each update is stamped with the server and time at creation. Each server maintains a list describing the time-stamp of the lastest updates it has seen from every other server. By referring to this list, exactly the necessary updates are exchanged.

It is necessary that a deletion update be kept for a period of time until it has had a chance to propagate to all sites. Using the time-stamp on the update, Deceit automatically discards deletion updates after 5 minutes to prevent unbounded use of memory.

Using a Monte Carlo simulation, we have found that in a network of 500 servers it takes an average of less than 70 seconds to completely propagate[8] an update. There is a negligible probability that the propagation will take more than 90 seconds. Moreover, this protocol allows updates to be efficiently collected into large batches, keeping the communication overhead low. Also, epidemic propagation is also highly crash resilient, since a server automatically learns about all missed updates when it recovers from a crash

3.2.5 File Behavior Control Parameters

The communication and replication protocols are controlled by several file parameters which can be varied on an individual file basis. The main file parameters are these:

1. Minimum Replica Level - the minimum number of valid and available replicas that must be maintained. For example, a minimum replica level of 3 would force Deceit to always maintain a valid replica on at least 3 separate servers. By default, the value is 1.

2. Write Safety Level - the number of replica servers that must reply to an update before a *write* RPC returns to a client. A value of 0 produces asynchronous unsafe writes; a value greater than or equal to the number of available replicas produces slow and fully synchronous writes. By default, the value is 1.

3. Stability Notification - specifies whether stability notification is to be used (as described in Section 3.2.3). Stability notification is usually invisible to applications. It may be turned off by a user if the performance cost is too high. The default is to use stability notification.

4. File Migration - specifies whether Deceit will automatically attempt to migrate a file to servers that receive requests for that file. File migration is accomplished by creating a replica on one server and deleting a replica on another; replica creation and deletion are atomic with respect to updates. File migration provides improved read performance and improved resilience to network failures. For some applications, it may be bad to automatically generate local replicas. For example, a backup log file is costly to copy, and it is rarely read. The default is that file migration not be used.

5. Write Availability Level - determine when Deceit generates a new write-token. If this flag is set to "high," then a server generates a new token whenever a token can not be located. A high availability makes it possible that inconsistent file replicas will result due to a partition. (The resolution of conflicting replicas is discussed in Section 3.3.1.) A value of "medium" allows a server to generate a new token or use an old token only when it can contact a majority of the replicas. As a result, some replicas may occasionally be "read only," but inconsistent file replicas will occur much less frequently. A value of "low" prevents the production of additional tokens. Loss of file write access may be more frequent and outages will last longer, but there is no chance of generating inconsistent replicas. The default value is "low."

Note that when the default values are used, a file behaves like a vanilla NFS file.

3.3 Fault Tolerance

The main effect of crashes and network partitions is that replicas (and caches) can become inconsistent. The two main mechanisms for dealing with this problem are the epidemic hard link propagation algorithm described in Section 3.2.4, and the file version control mechanism discussed here.

3.3.1 File Versions

Deceit maintains detailed version information about every replica of each file. This information is sufficient to quickly determine if two replicas are identical without actually comparing the data. Deceit does not explicitly store the full list of updates for a replica. Instead, there is a one-to-one mapping from replicas to integer pairs (v,w) where v is the *major version number,* and w is the *minor version number*[9]. w is incremented on every update, and v is changed to a new (and unique) number every time there is a potential divergence between replicas. In practice, a major version number is generated every time a new token is generated. As long as a single write-token controls a set of replicas, the minor version number completely describes the state

[8] Since information exchange is not simultaneous everywhere, an update may experience multiple propagations within the same "round."

[9] In the literature, this value is often called an *update counter.*

of each replica.

A version pair is stored with each write-token. The token version pair can be compared to a replica version pair to quickly decide if a replica has received every update "through" that token. Additionally, the version pairs associated with two replicas can be compared to determine if the two replicas are identical. These version pairs are also available to the user through a special command so that the user can determine the state of a file.

When a new major version is generated, the user is notified through the use of a special log file. It is then the user's responsibility to merge file versions. This is similar to the Locus and Coda file system[24]. Note that if the file availability level is set to "medium," it should be rare that multiple file versions are produced.

Users can directly access specific file versions using normal UNIX operations by qualifying files names using a special syntax. This mechanism may also be used as a normal file versioning system, such as in a source code management system. For example, major version 3 of "foo" can be referred to as "foo;3." By using this form of file name, specific versions can be modified and deleted. Deceit interprets these special files names; there is no actual file called "foo;3." By using an unqualified file name, the user automatically refers to the most recent available version.

3.3.2 Crash Recovery

ISIS provides a clean model of site failure and recovery. If a site appears to fail to any server, then it appears to fail to all servers. Communication is reliable, and all membership changes appear to be atomic with respect to communication, i.e. a broadcast occurs entirely before or after a failure or recovery. In order to explain the usage of the crash resilience mechanisms is Deceit, we present several scenarios.

Non-token Replica Crash

When a server *s* recovers from a crash, *s* contacts the token holder for each file *f* such that *s* has a replica but no token for *f*. Each token carries the version pair that replicas should have if they are up to date. If *s* finds that it has an obsolete replica of *f*, *s* destroys the obsolete replica.

Token Crash

The crash of a server holding the token is detected when a server attempts to contact the token holder during the course of normal *read* or *write* operations. Assume that server *s* needs to distribute an update for a file, but it cannot contact the current token holder. Subject to token generation constraints, *s* generates a new token. Since *s* now holds the token for a version of the file, *s* can complete the original operation.

When the crashed server recovers, it will be notified about the creation of the new version during its recovery. This server will note that the new version is a direct descendent of the old version, and it will destroy the old version and all of its replicas.

Partition

Now consider the scenario where there was a network partition, but no updates were issued to the file in the partition containing the token. *Read* access on both sides continues normally, since it is difficult distinguish between this scenario and the case where the other replicas simply crashed. *Write* access in the partition which does not contain the old token may cause a new token to be generated. When the partition is resolved, the old token holder will be notified. It will appear to the clients as if the token had actually been moved, and the updates were propagated very slowly to some servers.

The hard case is when a partition occurs and updates are issued to the file on both sides concurrently. In this case both of the incomparable versions of the file are kept, and a notification is given to the user using a log file, as was discussed in Section 3.3.1.

Stability Notification in the Presence of Failure

If the token holder *t* for a file *f* loses contact with some of *f's* replicas during an update distribution, then those replicas might be left in an inconsistent state. Stability notification is used to detect this case. Before an update is distributed, all available replicas are marked as *unstable.* Therefore, if replica states are inconsistent, then all inconsistent replicas will be marked as *unstable.*

Inconsistency is detected when a *read* is given to a server *s* that has an *unstable* replica of *f*, and *s* is unable to contact *t*. In order to respond to a *read, s* must locate a *stable* replica by broadcasting to all replicas of *f* to determine their state. If there is a *stable* replica at server *s'*, the operation is forwarded to *s'*. If no replica is marked as *stable, s* forces the most up to date replica to be *stable* and all obsolete replicas are destroyed.

Hard Link Recovery

Since the epidemic broadcast protocol is relatively slow, it is unlikely that all the servers will be consistent during a crash. Further, a disk log of updates is kept, but for performance reasons, this log is written synchronously only for local creations and deletions. Other updates are buffered and written asynchronously to reduce overhead.

As a result, a recovering server will not have an accurate list of all hard links. If any other server is running, that server is contacted to determine the current list. To recover from a failure at all sites, the disk log at all sites must be merged[13].

3.4 Scalability

The goal of scalability in Deceit is to provide as many cooperating servers as possible without suffering an overburdening performance overhead. Some system changes, such as server crashes, need to be broadcast to all sites, and their frequency will increase linearly with the number of sites. Therefore, there is always a per server overhead of at least $O(n)$ associated with a closely coupled group of n servers. In order to minimize this problem, Deceit uses the following principle: **each server stores the minimum amount of information necessary to accomplish its work.**

File Groups

A basic mechanism for controlling knowledge about a file is the *file group.* For any file, *f,* there is an explicit set of servers, or process group, that need current information about *f,* which we will call the *file group* of *f.* There must be a mechanism for broadcasting messages to all of a file group members and sending messages to individual members. Deceit uses ISIS facilities to efficiently represent and manipulate the membership of a file group.

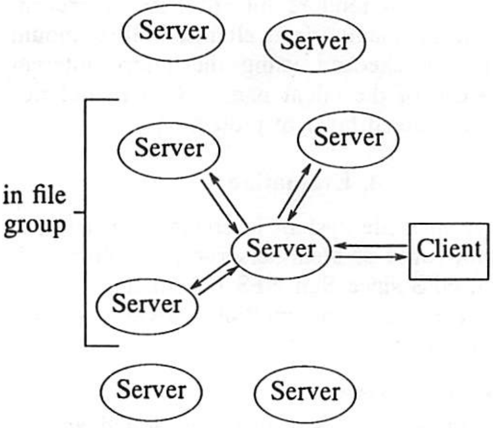


Figure 3 - Update Distribution in a Group

The file group for *f* contains all servers that have a replica of the file or have cached information about the file. This set is a superset of the replica holders, and it includes those servers which cache only timestamps or mode bits. The fundamental operation within a file group is *update distribution.* An update to *f* originates from a client and is given to its server. That server then acquires the write-token and broadcasts the update to all members of *f's* file group; no other servers receive this update for *f.* Refer to Figure 3 for a schematic description of update distribution. The concept of a file group is fundamental to the scalability of the entire system, since only the size of *f's* file group affects the speed of updates to *f.*

Cells

Another mechanism which provides scalability is *cells.* In the above discussion, it was assumed that all clients could directly access any Deceit server, but this property is not necessarily true. Sets of Deceit servers can be subdivided into *cells* to help security and efficiency in Deceit in a very large implementation. Each cell is an independent instantiation of Deceit with distinct files and processes. Each cell maintains its own name space, and replication must be contained within a cell. A cell provides security and administrative boundaries. In our present implementation, cells correspond to ISIS site clusters. An example of Deceit cells is shown in Figure 4.

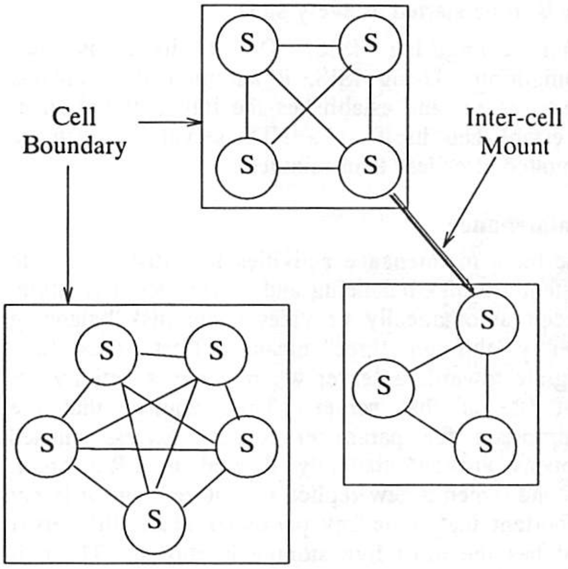


Figure 4 - Deceit Cells

Access between cells is provided through a logical directory. There is a logical directory called the *global root directory.* It cannot be listed, as it implicitly contains the full host names of every accessible Deceit server. Instead, it is used indirectly as a subdirectory of a normal directory. For example, if a user in the Cornell computer science cell wants to access files in the CMU computer science cell, he picks a machine **foo.cs.cmu.edu** at CMU where a Deceit server is running. By executing the command ''cd /priv/global/foo.cs.cmu.edu,'' a user can access the CMU cell with normal file operations. The global root directory is a subdirectory of **/priv.** The Cornell cell acts as a NFS client to the CMU cell. Mount and access restrictions are applied as with any client.

3.5 Administration

The administration of Deceit can be divided into three components: initialization, maintenance, and security. Since Deceit is not yet in daily use at the time that is paper is being prepared, these issues will not be presented in the detail that would result from real experience.

Initialization

The bulk of initialization beyond normal NFS client preparation is in the configuration of ISIS. A **sites** file must be prepared listing all server sites by host names. Additional names can be added later. This file provides all the information that ISIS needs so that servers can contact each other. After preparing this file and compiling ISIS, and an ISIS server needs to be started at every site.

After configuring ISIS, Deceit itself is self configuring. Using ISIS, it automatically contacts other servers and establishes the initial global state. It establishes itself as a NFS server and can be mounted after less than minute.

Maintenance

The main maintenance activities in a distributed file system are disk balancing and server reconfiguration. Deceit automatically provides some disk balancing silently through three means. First, files may migrate towards a server where there is activity for that file at that server. This requires that the appropriate file parameter is set. Excess unused replicas are automatically deleted in LRU order. Second, when a new replica is required, and it is not important that it be any particular place, the server that has the most free storage is chosen. Third, if server disk utilization passes a high water mark, replicas are automatically moved to other servers in background.

Server reconfiguration breaks down into several cases:

1. Temporarily removing a server - this is an easy case since Deceit must be tolerant of crashes anyway. Asking Deceit to quit takes about 5 seconds: it has to close all of its local files. ISIS will broadcast the shutdown to all other servers as a site failure.

2. Long term removal of a server - this case is more difficult since the server may have a large number of local file replicas. First, all replicas need to be forced to other servers. Then, Deceit can be shut down as in case 1.

3. Adding a server at a site which is in the ISIS **sites** file - again this is an easy case. Starting up Deceit will cause it to automatically configure itself. Deceit will immediately begin accepting file replicas from other sites and allowing clients to mount.

4. Adding a totally new server - adding a site name to the ISIS **sites** file requires that ISIS, and Deceit, be shutdown and restarted. This can be accomplished one server at a time at all sites, but it is still disruptive. After accomplishing this, Deceit can be started as in case 3.

Reconfiguring a client is much easier than in NFS. In normal NFS, each client needs to have a list of every server in its **/usr/fstab** file. For a large installation, this restriction becomes a major inconvenience. In Deceit, each client only needs to list one Deceit server to get full service. Adding a server requires no activity at existing clients except to move clients to balance server load.

Security

Deceit does not directly address most security issues. It is assumed that communication between instances of the server is secure (e.g. encrypted or physically secure). Also, the local files used for storage by Deceit are assumed to be inaccessible to unauthorized users. Client/server communication is secured, and client authentication is provided using some other mechanism. It is beyond the scope of this paper to provide a detailed description of these mechanisms.

Deceit does provide a few security features. File handles contain 64 random bits of data to prevent forgery from an unauthorized clients. Client mount authorization is checked using the return internet address instead of the client name. Cell boundaries provide an additional layer of protection.

4. Evaluation

Performance in a file system is crucial. To analyze Deceit performance as a single server, we compare it against Sun NFS since Sun NFS is a mature, optimized file server. For the multiple server case, we will analyze Deceit by itself.

Single Server Analysis

Table 2[10] compares NFS against Deceit indexed by RPC type. Refer to the appendix for a description of the RPC types. All times are in milliseconds. The number in parenthesis is the operation size in bytes.

Of the RPC types, WRITE, CREATE, and REMOVE seem to be the most inefficient. First we analyze WRITE. The fact that Deceit performs as well for 1 byte WRITE calls as NFS is indicative of the fact the Deceit is slower because of additional data copying. It take approximately 5 ms to copy 8000 bytes on a Sparcstation, therefore Deceit must be doing 3 more data copies than Sun NFS. Deceit is at a disadvantage since it runs at the user level as opposed to NFS which runs in the kernel. Figure 5 illustrates the work that must be accomplished by a

[10] These times were obtained by using a dedicated Sparcstation as a client and a separate one as a server. The software issued the Sun RPC calls directly to avoid the overhead of going through the client file system interface.

user level server daemon for a WRITE RPC.

RPC type	NFS time	Deceit time	Ratio
NULL	2.54 ms	3.19 ms	1.26
GETATTR	3.32	4.00	1.20
LOOKUP	6.24	4.92	0.79
READDIR	7.56	10.5	1.39
READ(1)	3.92	4.31	1.10
READ(8000)	17.6	20.9	1.19
WRITE(1)	49.5	49.5	1.00
WRITE(8000)	66.7	82.6	1.24
CREATE	61.8	162	2.62
REMOVE	17.0	60.4	3.55
RENAME	79.2	46.6	0.59
SETATTR	3.72	51.7	13.9
MKDIR	139	75.1	0.54
RMDIR	60.3	63.0	1.05
STATFS	3.15	4.01	1.27

Table 2 - Performance of NFS vs. Deceit

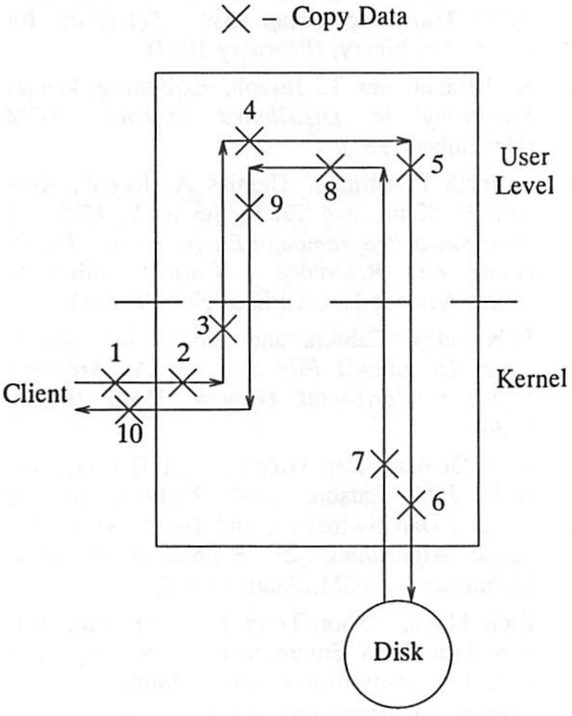


Figure 5 - Control Path in Server

In step 1, the WRITE RPC and its data is copied off of the ethernet device and into a kernel buffer. This usually requires assembling the message causing copy number 2. In step 3, the message is copied into user memory. In step 4, the WRITE parameters are broken out of the message into machine dependent form. In the standard Sun RPC library, the data is always copied at this step. In step 5, the data is copied back into the kernel in the course of a UNIX "write" call. In step 6, the data is copied to the disk device. In step 7, the return value of the write is copied into user memory. In step 8, the reply message is prepared. In step 9, the reply message is copied back down to the kernel. Finally, in step 10, the reply message is copied to the ethernet device. All told, the file data is copied 6 times. The three extra copies mentioned above correspond to steps 3 and 5 plus one one other copy that can be optimized out within the kernel. We believe that Deceit writes can not be made substantially faster unless either Deceit is moved into the kernel or UNIX provides a more efficient network and disk interface.

CREATEs and REMOVEs are expensive because, in the current implementation, an ISIS process group must be created or deleted respectively. In order to maintain communication atomicity, ISIS must temporarily stop communication at all servers until the creation or deletion is complete. This stoppage involves expensive network communication and process blocking. We are presently making structural changes to Deceit to eliminate this cost.

Multiple Server Analysis

We tested the total throughput for a serial stream of writes to multiple destinations. All writes were issued through one server so that the write-token did not have to move around. The results are summarized in Table 3.

	Size of Data	
Number of Replicas	1	8000
1	49.5	82.6
2	83.1	134
3	83.2	157
4	88	247

Table 3 - Multiple Destination Write Times

The sharp break between one replica and two replicas is because Deceit uses an optimization when a file group has only one member. If this optimization is not used, Deceit takes 83 ms to accomplish a write of 1 byte for a single replica. The source of this overhead is that we used version 1.3.1 of ISIS which requires that all broadcasts go through the ISIS server process. The current version of ISIS, version 2.0, does not have this limitation, but it is not stable enough at the current time.

Since Deceit performed nearly as well for 4 replicas as it did for 2 when the writes were small, it is clear that Deceit is making good use of concurrency. The poor performance of Deceit when their were 4 replicas and large writes is due to bottlenecks in the UNIX network software and ISIS. Both suffer catastrophic degradation when internal buffers begin to overflow.

5. Conclusion

Most of the distributed file systems that are in use today were developed for and characterized by relatively small files and applications similar to program development. This fact is not surprising, since program development has been one of the most prevalent distributed applications. New file systems need to be developed that will support a much wider variety of applications and scale to very large networks. Currently, such new applications would require customized solutions designed and implemented individually.

Deceit addresses many of these concerns. With Deceit, the user can specify some details about how a file is to be managed. We have chosen a set of properties that we believe are important, as they allow tradeoffs on reliability, read performance, write performance, and consistency. The value of our choices, however, must be tested.

We have built a prototype of Deceit, and are now evaluating it. There have been several major revisions in the design to date, and more changes are inevitable. Nonetheless, Deceit appears to be useful and efficient. By the time of this conference we hope to be using Deceit in our normal daily operations.

Appendix

The Sun Network File System[1] is a client/server protocol based on remote procedure calls (RPCs). A client forms a request into a network message and sends it to the server. Exactly one reply is sent back to the client. File and directories are specified using *file handles.* File handles are 32 byte blocks of data which identify a file or directory to the server. The list of RPC types is as follows:

NULL - no operation. This call is used to test the minimum response time of the server.

GETATTR - get all file attributes

SETATTR - set some file attributes. These attributes include owner id and permission bits.

LOOKUP - get a file handle using the parent directory and a name

READDIR - list all names in a directory

READ - read a block of data from a file. A block of data can be read starting at any position but no more than 8 kilobytes can be read with one call.

WRITE - write a block of data

CREATE - create a file

LINK - add a hard link to a file

REMOVE - destroy a hard link to a file. If there are no more hard links to the file, it is deallocated.

RENAME - rename/move a file

SYMLINK - create a soft link to a file

READLINK - read a soft link

MKDIR - create a directory

RMDIR - destroy a directory. Only empty directories may be destroyed.

STATFS - get file system statistics

NFS servers are *stateless.* That is, they do not have to store any dynamic information about clients. This fact allows NFS servers to recover from a crash easily without contacting other machines.

References

1. Anonymous, *Network File System Protocol Specification,* Sun Microsystems Inc., Mountain View CA (February 1986).
2. Kenneth Birman and Thomas Joseph, "Reliable Communication in the Presence of Failures," *ACM Trans. of Comp. Sys.* 5(1)Assn. for Comp. Machinery, (February 1987).
3. K. Birman and T. Joseph, *Exploiting Virtual Synchrony in Distributed Systems,* ACM (November 1987).
4. Kenneth P. Birman, Thomas A. Joseph, Kenneth P. Kane, and Frank Schmuck, *ISIS - A Distributed Programming Environment - User's Guide and Reference* , Cornell University Comp. Science Dept., Ithaca New York ().
5. Luis Felipe Cabrera and Jim Wyllie, *QuickSilver Distributed File Services: An Architecture for Horizontal Growth,* IEEE (March 1988).
6. Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry, *Epidemic Algorithms for Replicated Database Maintenance,* ACM (August 1987).
7. Rick Floyd, "Short-Term File Reference Patterns in a UNIX Environment," Tech. Rep. No. 177, University of Rochester (March 1986).
8. Rick Floyd, "Directory Reference Patterns in a UNIX Environment," Tech. Rep. No. 179, University of Rochester (August 1986).
9. Robert Joseph Fowler, *The Complexity of Using Forwarding Addresses for Decentralized Object Finding,* ACM (August 1986).
10. Robert Hagmann and Group Commit, *Reimplementing the Cedar File System Using Logging,* ACM (November 1987).
11. John H. Howard, Michael L. Kazar, Sherri G. Menees, Davic A. Nichols, M. Satyanarayanan, Robert N Sidebotham, and Michael J. West, "Scale and Performance in a Distributed File System," *ACM Trans. of Comp. Sys.* 6(1) pp.

51-81 Assn. for Comp. Machinery, (February 1988).

12. Thomas A. Joseph and Kenneth P. Birman, "Low Cost Management of Replicated Data in Fault-Tolerant Distributed Systems," *ACM Trans. of Comp. Sys.* 4(1)Assn. for Comp. Machinery, (February 1986).
13. Kenneth Kane, "Log-Based Recovery in Asynchronous Distributed Systems," Thesis (January 1990).
14. Gerard LeLann, *Distributed Systems - Architecture and Implementation An Advanced Course,* Springer-Verlag (September 1981).
15. Barbara Liskov and Rivka Ladin, *Highly Available Distributed Services and Fault-Tolerant Distributed Garbage Collection,* ACM (August 1986).
16. Imothy Mann, Andy Hisgen, and Garret Swart, "An Algorithm for Data Replication," Tech. Rep. No. 46, Digital Equipment Corporation Sys. Research Center (June 1989).
17. Keith Marzullo and Frank Schmuck, "Supplying High Availability with a Standard Network File System ," Tech. Rep. No. 87-888, Dept. of Comp. Science at Cornell University (December 1987).
18. James G. Mitchell and Jeremy Dion, *A Comparison of Two Network-Based File Servers,* ACM (December 1981).
19. Sape J. Mullender, *A Distributed File Service Based on Optimistic Concurrency Control,* ACM (December 1985).
20. Michael N. Nelson, Brent B. Welch, and John K. Ousterhout, "Caching in the Sprite Network File System," *ACM Trans. of Comp. Sys.* 6(1)Assn. for Comp. Machinery, (February 1988).
21. John K. Ousterhout, Herve Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson, *A Trace-Driven Analysis of the UNIX 4.2 BSD File System,* ACM (December 1985).
22. M. Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere, "Coda: A Highly Available File System for a Distributed Workstation Environment," Tech. Rep. No. CMU-CS-89-165, Dept. of Comp. Science at Carnegie Mellon University (November 1989).
23. Carl Staelin, "File Access Patterns," Tech. Rep. No. CS-TR-179-88, Dept. of Comp. Science at Princeton University (September 1988).
24. Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel, *The LOCUS Distributed Operating System,* ACM (October 1983).

Alex Siegel graduated from Rice university with a B.S. in Electrical Engineering after the Spring of 1987. He then became a graduate student of the Computer Science Department at Cornell University. His current research is the Deceit distributed file system, a file system that provides high reliability and flexibility. His advisor is Keith Marzullo.

Kenneth P. Birman received his Ph.D. in Computer Science from U.C. Berkeley in 1981, and is currently Associate Professor in the Department of Computer Science at Cornell University. He heads the ISIS Project, which has developed a toolkit for reliable distributed computing, and is founder and President of ISIS Distributed Systems Inc., an Ithaca-based consulting company that is developing fault-tolerant distributed software and program management aids.

Keith Marzullo received his Ph.D. from Stanford University in 1984, spent two years at the Xerox Corporation, and then joined the faculty of the Computer Science Department at Cornell University in 1986. His recent research has been into the principles of reliable reactive systems. He heads the Meta project, which is an implementation effort to develop new technologies that support the development of reliable reactive systems, both with and without real-time constraints. He has also worked on fault-tolerant file systems, distributed software manufacturing systems, and on distributed clock synchronization.

USENIX Summer Conference
June 11-15, 1990
Anaheim, California

Implementation of the Ficus Replicated File System

Richard G. Guy, John S. Heidemann, Wai Mak, Thomas W. Page, Jr., Gerald J. Popek, Dieter Rothmeier – University of California Los Angeles

ABSTRACT

As we approach nation-wide integration of computer systems, it is clear that file replication will play a key role, both to improve data availability in the face of failures, and to improve performance by locating data near where it will be used. We expect that future file systems will have an extensible, modular structure in which features such as replication can be "slipped in" as a transparent layer in a stackable layered architecture. We introduce the Ficus replicated file system for NFS and show how it is layered on top of existing file systems.[1]

The Ficus file system differs from previous file replication services in that it permits update during network partition if any copy of a file is accessible. File and directory updates are automatically propagated to accessible replicas. Conflicting updates to directories are detected and automatically repaired; conflicting updates to ordinary files are detected and reported to the owner. The frequency of communications outages rendering inaccessible some replicas in a large scale network and the relative rarity of conflicting updates make this optimistic scheme attractive.

Stackable layers facilitate the addition of new features to an existing file system without reimplementing existing functions. This is done in a manner analogous to object-oriented programming with inheritance. By structuring the file system as a stack of modules, each with the same interface, modules which augment existing services can be added transparently. This paper describes the implementation of the Ficus file system using the layered architecture.

Introduction

The Ficus project at UCLA is investigating very large scale distributed file systems. We envision a transparent, reliable, distributed file system encompassing a million hosts geographically dispersed across the continent, perhaps around the globe. Any host should be able to access any file in the distributed system with the ease that local files are accessed.

A large scale distributed system displays several critical characteristics: it is subject to continual partial operation, global state information is difficult to maintain, and heterogeneity exists at several levels. A successful large scale distributed file system must minimize the difficulties that these characteristics imply.

The scale of such a distributed system implies that the system will never be fully operational at any given time. For a variety of technical, economic, and administrative reasons various system components such as hosts, network links, and gateways will at times be unusable. Partial operation is the normal, not exceptional, status of this environment; new approaches are needed to provide highly available services to such a system's clients.

Large scale also prevents most nodes from attempting to maintain information about the global state of the system. (Imagine a filesystem table with millions of entries.) Even hosts with sufficient storage resources can not effectively track all the changes that occur across the distributed system, either because the changes are too rapid or communication is unreliable between the source of the change and the monitor.

Very large scale also implies that a high degree of hardware, software, and administrative heterogeneity exists. Software that can provide the desired availability must be easily utilized by a wide variety of existing host environments; it must also be tunable to meet both technical and administrative concerns. New tools must be sufficiently modular to allow easy attachment to existing services, and yet still provide acceptable performance.

These issues led us to explore the application and integration of several concepts to large scale file systems: stackable layers, file usage locality, data

[1] This work was sponsored by DARPA under contract number F29601-87-C-0072.

replication, non-serializable consistency, and dynamic volume locating and grafting.

Stackable layers: The stackable layers paradigm is used by Ritchie [16] as a model for implementing the stream I/O service in System V UNIX[2]. A stackable layer is a module with symmetric interfaces: the syntactic interface used to export services provided by a particular module is the same interface used by that module to access services provided by other modules in the stack. A stack of modules with the same interface can be constructed dynamically according to the particular set of services desired for a specific calling sequence.

We have found this model to be useful in designing and constructing file systems, as it allows easy insertion of additional layers providing new services. We have used it to provide file distribution and replication; we expect to use it for performance monitoring, user authentication and encryption.

File usage locality: The importance of modularity and portability implied that our replication service build on top of the existing UNIX file system interface. At least one previous attempt to adopt this philosophy abandoned it in the face of poor performance [19] general purpose (university) UNIX file usage [6, 5] indicate a strong degree of file reference locality, and that appropriate caching methodologies can exploit this behavior to reduce file access overhead. The Ficus file system design takes advantage of these locality observations to avoid much of the overhead previously encountered in building on top of an existing UNIX file system implementation.

Replication: Data replication is used to combat the partial operation behavior that tends to degrade availability in large scale file systems. Each host may store one or more physical replicas of a logical file; clients are generally unaware which replica services a file request. The replication techniques used in Ficus are intellectual descendants of those used in the Locus [15] distributed operating system.

Non-serializable consistency: Most data replication management policies proposed in the literature adopt some form of serializability as the definition of correctness. The requisite mutual exclusion techniques to enforce serializability typically display an inverse relationship between update availability and read availability: ensuring high read availability forces a low update availability.

Ficus incorporates a novel, non-serializable correctness policy, *one-copy availability*, which allows update of any copy of the data without requiring a particular copy or a minimum number of copies to be accessible. One-copy availability is used in conjunction with automatic update propagation and directory reconciliation mechanisms to ensure (or promote) consistency of all replicas.

One-copy availability provides strictly greater availability than primary copy [2], voting [21], weighted voting [7], and quorum consensus [10]. Our directory reconciliation mechanism tolerates a larger class of concurrent non-serializable updates than the replicated ''dictionaries'' of [4, 1, 22] The replicated directory techniques in [3,18] are based on quorum consensus, and thus also have lower availability. The Deceit file system [20] allows partitioned update without a quorum, but has no mechanism for reconciling concurrent updates to replicas of a single directory.

Volume locating and grafting: Locating a particular file in a very large scale distributed system requires a robust, distributed mechanism. Dynamic movement of files must be supported without requiring any sort of advance global agreement. Ficus incorporates a volume autograft mechanism along with a segmented, distributed, replicated graft table.

The remainder of this paper describes key architectural details of the Ficus file system as of April, 1990. Further discussion of the ideas touched on above can be found in [13,9,8]

Ficus layered design

The Ficus layered file system model comprises two separate layers constructed using the vnode interface. NFS is employed as a transport mechanism between remotely located Ficus layers, and can also be used as a means for non-Ficus hosts to access Ficus file systems. Figure 1 shows the general organization of Ficus layers; the NFS layer is omitted when both layers are co-resident.

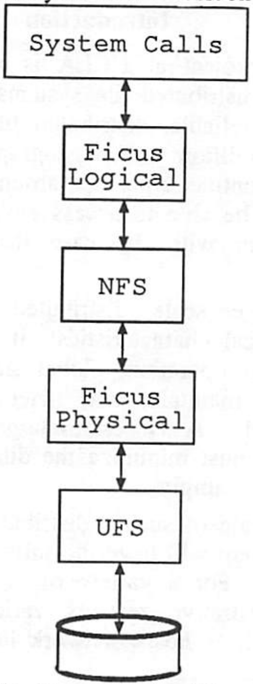


Figure 1: Ficus Stack of Layers

Vnode interface

The single most important design decision to be made when using the stackable layers paradigm is the definition of the interface between layers. Ideally, the interface will be general enough to allow for later extensibility in unplanned directions. The *streams* interface [16], for example, is remarkably simple and general: messages may be placed on an input queue for processing by the layer. Each layer dequeues and processes messages of types it recognizes; unrecognized message types are passed on to the next layer in the sequence.

Interface definitions can also be more closely tailored to the particular application area, as is the case with the vnode [12] interface used in SunOS for file system management. The vnode interface is defined by a set of about two dozen services, together with their calling syntax and parameters. In SunOS, the vnode interface is used to hide details of particular file system implementations, including the location (local or remote) of the actual file storage.

We adopted the vnode interface for stackable layers in Ficus, with some misgivings. Leveraging an existing interface for file system modules is clearly beneficial when getting started. The vnode interface is also in widespread use, so persuading others to add Ficus modules to existing implementations is much easier than introducing an entirely new interface. On the other hand, the vnode interface is quite rigid: adding services desired by new layers encountered a variety of difficulties, of which several are mentioned below.

Using the vnode interface also allows Ficus to utilize existing UFS (UNIX File System) and NFS (Network File System) [17] services in SunOS in critical ways. For example, Ficus can use the UFS as its underlying nonvolatile storage service, which means Ficus is not burdened with the details of how best to physically organize disk storage. Ficus is also able to use NFS as its remote access and transport mechanism, again relieving Ficus of substantial work.

While the Ficus layers are conceptually organized as in Figure 1, each is implemented as a new virtual file system type, as indicated in Figure 2.

NFS as a transport layer

NFS is essentially a host-to-host transport service with a vnode interface. Generally speaking, then, any layer that uses a vnode interface can be unaware whether the immediately adjacent functional layers are local, or remote and accessed via an intervening NFS layer. The Ficus replication service layers are able to use NFS for transparent access to remote layers, without having to build a transport service.

Unfortunately, the NFS implementation in SunOS does not fully preserve vnode semantics. The stateless philosophy of NFS clashes occasionally with vnode semantics, and the resulting NFS implementation is not simply a "host-to-host transport service with a vnode interface". For example, the vnode services `open` and `close` are not supported by the NFS definition, and so are ignored: a layer intending to receive an `open` will never get it if NFS is in between.

NFS also incorporates optimizations intended to reduce communications and improve performance. The file block caching and directory name lookup caching are not fully controllable (e.g., there is no

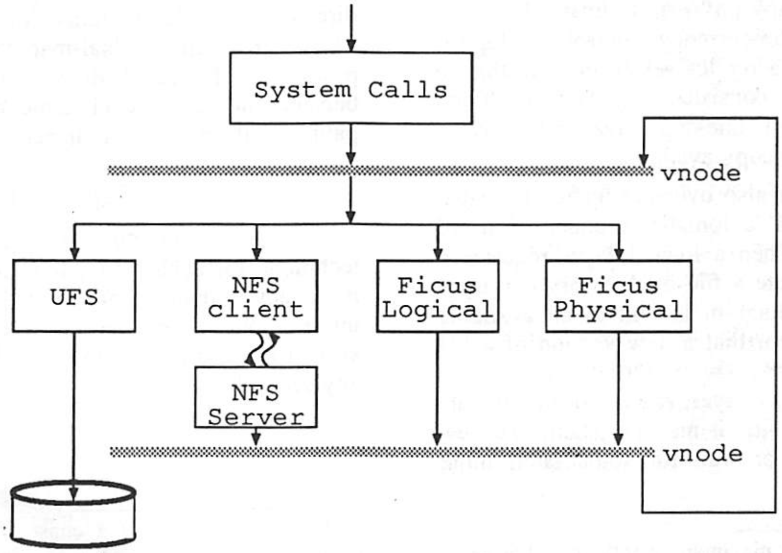


Figure 2: Layered Architecture Using Vnodes

user-level way to disable all caching), which results in unexpected behavior for layers which are not able to adopt the assumptions inherent in the NFS cache management policies.

Adding new vnode services

The Ficus replication service employs functionality not anticipated (understandably) by the vnode interface design. Rather than add several new services outside the vnode framework (as in Deceit [20]) we chose to overload existing vnode services. This maximizes portability, at a slight expense of interpreting an overloaded service and perhaps limiting its use in some way.

For example, Ficus is able to use effectively the open/close information that NFS intercepts and ignores, so a new service is required. We overloaded the `lookup` service by encoding an open/close request as a null-terminated ASCII string of sufficient length to be passed on by NFS without interpretation or interference.[3]

Cooperating layers

Layers can be added to a stackable design singly or in groups. Layers inserted as a group may be stacked together or separated by other, existing layers. For example, the Ficus replication service is composed of two layers, a *logical file layer* and a *physical replica layer* are separated by an NFS layer when the logical and physical layers are on different hosts.

Ficus Logical layer

The Ficus logical layer presents its clients (normally the UNIX system call family) with the abstraction that each file has only a single copy, although it may actually have many physical replicas. The logical layer performs concurrency control on logical files and implements a replica selection algorithm in accordance with the consistency policy in effect. The default policy of one-copy availability is to select the most recent copy available.

The logical layer also oversees update propagation notification and automatic reconciliation of directory replicas. When a logical layer requests a physical layer to update a file or directory, an asynchronous multicast datagram is sent to all available replicas informing them that a new version of a file may be obtained from the replica receiving the update. Each physical layer reacts to the update notification as it sees fit: it may propagate the new version immediately, or wait for some later, more convenient time.

Periodically, a logical layer invokes a file and directory reconciliation mechanism to compare file replica subtrees. The details of the reconciliation algorithms are beyond the scope of this paper; see [9, 8] for further information.

Ficus files are organized in a general DAG of directories; unlike UNIX, Ficus directories may have more than one name.[4] The logical layer maps a client-supplied name into a Ficus file handle, which contains a set of fields that uniquely identify the file across all Ficus systems. The Ficus file handle is used to communicate file identity between the logical and physical layers.

Ficus Physical layer

The Ficus physical layer implements the concept of a file replica. Each Ficus file replica is stored as a UFS file, with additional replication-related attributes stored in an auxiliary file. (These attributes would be placed in the inode if we were to modify the UFS.) Ficus uses the version vector technique of [14] to detect concurrent unsynchronized updates to files.

Ficus directories are stored as UFS files, not UFS directories. A Ficus directory entry maps a client-specified name into a Ficus file handle, which then must be mapped into an inode by the UFS. This second mapping is implemented by encoding the Ficus file handle into a hexadecimal string used by the UFS as a pathname.

The dual-mapping nature of the current Ficus implementation is difficult to implement efficiently [19], but is not inherently expensive. The on-disk file organization closely parallels the logical Ficus name space topology, which allows the existing UFS caching mechanisms to continue to exploit the strong directory and file reference locality observed in [6, 5] [19] in a similar dual-mapping scheme used in a prototype of the Andrew File System occurred because the lower level name mapping was incompatible with the locality displayed at higher levels.

Replication

Ficus incorporates data replication as a primary technique for achieving a high degree of availability in an environment characterized by communications interruptions. Each file and directory in a Ficus file system may be replicated, with the replicas placed at any set of Ficus hosts.

[3] The reduction in the maximum length of a file name component from 255 to about 200 does not seem to be a significant loss: we've never seen a component of even length 40.

[4] This characteristic is a consequence of the ability to change the name of a directory while some copies are unavailable. When non-communicating directory replicas are concurrently given new names, it is often later necessary to retain multiple names.

Basics

A logical file is represented by a set of physical replicas. Each replica bears a *file identifier* that globally uniquely identifies the logical file, and a *replica identifier* that uniquely identifies that particular replica. The logical layer uses a file handle composed (in part) of file identifier and replica identifier to communicate with physical layers about a file.

The number and placement of file replicas is effectively unbounded.[5] A client may change the location and quantity of file replicas whenever a file replica is available.

Associated with each file replica is a *version vector* [14] which encodes the update history of the replica. Version vectors are used to support concurrent, unsynchronized updates to file replicas managed by noncommunicating physical layers.

Update notification/propagation

Updates are initially applied to a single physical replica. The invoking logical layer notifies other physical layers managing replicas of the updated file that a newer version exists in the updated replica. A physical layer that receives an update notification makes an entry for the file in a *new version cache.* An update propagation daemon consults this cache to see what new replica versions require propagation, and performs this propagation when it deems it appropriate to expend the effort. Rapid propagation enhances the availability of the new version of the file; delayed propagation may reduce the overall propagation cost when updates are bursty.

For regular files, update propagation is simply a matter of atomically replacing the contents of the local replica with those of a newer version remote replica. Ficus contains a single-file atomic commit service to support file update propagation. A shadow file replica is used to hold the new version until it is completely propagated, and then the shadow atomically replaces the original by changing a low-level directory reference. If a crash occurs before the shadow substitution, the original replica is retained during recovery and the shadow discarded.[6]

Update propagation for directories is more difficult because of the side effects of directory update: files may be allocated, reference counts adjusted, and so on. Simply copying directory contents is incorrect; in a sense, a directory operation needs to be "replayed" at each replica. In Ficus, a *directory reconciliation* algorithm is used for this purpose.

Reconciliation

A reconciliation algorithm examines the state of two replicas, determines which operations have been performed on each, selects a set of operations to perform on the local replica which reflect previously unseen activity at the remote replica, and then applies those operations to the local replica.

The Ficus directory reconciliation algorithm [9] determines which entries have been added to or deleted from the remote replica, and applies appropriate entry insertion or deletion operations to the local replica. The standard set of UNIX directory operations is supported.

The directory reconciliation algorithm used for update propagation and the basic file update propagation service are both incorporated into the general Ficus file system reconciliation protocol. This protocol is executed periodically to traverse an entire subgraph (not just a single node), and reconcile the local replica against a remote replica. The execution proceeds concurrently with respect to normal file activity, so that client service is not blocked or impeded.

Volumes

Ficus uses volumes[7] as a basic structuring tool for managing disjoint portions of the file system. Ficus volume replicas are dynamically located and grafted (mounted) as needed, without global searching or broadcasting. The tables used for locating volume replicas are replicated objects similar to directories, and are managed by the same reconciliation algorithms used for directory replicas.

Basics

The Ficus file system[8] is organized as a directed acyclic graph of *volumes.* A volume is a logical collection of files that are managed collectively. Files within a volume typically share replication characteristics such as replica location and the number of replicas.

A volume is represented by a set of *volume replicas* which function as "containers" in which file replicas may be placed. The set of volume

[5] There is a current limit of 2^{32} replicas of a given file, and 2^{32} logical layers.

[6] Note that this commit service is not necessary for the correct operation of the general Ficus functionality. While its performance impact is usually small, it can have a significant effect if the client is updating a few points in a large file. To avoid alteration of the UFS, rewriting the entire file is necessary. That cost could, of course, be avoided by putting a commit function into the storage layer.

[7] Ficus volumes are similar to Andrew [11] volumes; both decouple the logical concept of subtree from the physical storage details in order to support flexible volume "replica" placement. Ficus does not require a replicated volume location database.

[8] We use *file system* (two words) to refer to a particular type of file service, e.g., UNIX file system or VMS file system. A *filesystem* (one word) is a self-contained portion of a UNIX file system normally one-to-one mapped into a single disk partition.

replicas forms a maximal, but extensible, collection of containers for file replicas. A volume replica may contain at most one replica of a file, but need not store a replica of any particular file.

A volume replica is stored entirely within a UNIX disk partition. The mapping between volume replicas and disk partitions is determined by the host providing the storage. Many volume replicas may be stored in a single partition; no relationship between volume replicas is implied by placement in disk partitions.

A volume is a self-contained[9] rooted directed acyclic graph of files and directories. A volume's boundaries are the root node at the top, and volume *graft points* at the bottom. The volume root is normally a directory; a graft point is a special kind of directory, as explained below. Each volume replica must store a replica of the root node; storage of all other file and directory replicas is optional.

Identifiers

A volume is uniquely named internally by a pair of identifiers: an *allocator-id*, and a *volume-id* issued by the allocator. Prior to system installation, each Ficus host is issued a unique value as its allocator-id; for example, an Internet host address would suffice. Individual volume replicas are further identified by their *replica-id*, so a volume replica is globally uniquely identified by the triple <allocator-id, volume-id, replica-id>.

Within the context of a particular volume, a logical file is uniquely identified by a *file-id*. A particular file replica is then identified by appending the replica-id of the containing volume replica to the file-id, as in <file-id, replica-id>. A fully specified identifier for a file replica is <allocator-id, volume-id, file-id, replica-id>; this identifier is unique across *all* Ficus hosts in existence.

Each volume replica assigns file identifiers to new files independently. To ensure that file-ids are uniquely issued, a file-id is prefixed with the issuing volume replica's replica-id. A file-id is actually, therefore, a tuple <replica-id, unique-id>.

Graft points

A graft point is a special file type used to indicate that a (specific) volume is to be transparently grafted at this point in the name space. Grafting is similar to UNIX filesystem mounting, but with a number of important differences. The particular volume to be grafted onto a graft point is fixed when the graft point is created, although the number and placement of volume replicas may be dynamically changed.

A graft point is very similar to a regular directory. It can be renamed or given multiple names. A graft point is itself replicated; a graft point replica is contained in a particular volume replica.

Many graft points for a particular volume may exist, even within a single volume. The resulting organization of volumes would then be a directed acyclic graph and not simply a tree.

A graft point contains a unique volume identifier and a list of volume replica and storage site address pairs. Therefore, a one-to-many mapping exists between a graft point replica and the volume replicas which can be grafted on it. Each graft point replica may have many volume replicas grafted at a time.

The list of volume replicas and the (Internet) addresses of the managing Ficus physical layers are conveniently maintained as directory entries. Overloading the directory concept in this way allows implicit use of the Ficus directory reconciliation mechanism to manage a replicated object (a graft point) with similar semantics and syntactic details.

Autografting

When the Ficus logical layer encounters a graft point while translating a pathname, a check is made to see if an appropriate volume replica is already grafted. If not, the information in the graft point is used to locate and graft the volume replica of interest.

A Ficus graft is very dynamic: a graft is implicitly maintained as long as a file within the grafted volume replica is being used. A graft that is no longer needed is quietly pruned at a later time.

Development methodology

The stackable layers paradigm extends to our development methodology. The vnode interface normally accessible only inside the kernel has been "exposed" to the application level through a set of *vnode system calls*, so that a functional layer can execute at the application level. The standard NFS server already provides a channel for a kernel layer to utilize a vnode layer in another address space; we customized a copy of the NFS server daemon code to run outside of the kernel as the interface to the Ficus layers.

This approach allows us to use application level software engineering tools to develop and test outside of the kernel what will ultimately be kernel level service layers. The performance penalty for crossing address space boundaries complicates performance measurements and analysis, but otherwise the methodology has proven sound.

The goal has been to provide a programming environment at the application level that is the same as a kernel-based module would experience. Today,

[9] Directory references (hard links) do not cross volume boundaries.

Ficus layers may be compiled for application level or kernel resident execution merely by setting a switch.

Our hope had been that once application level debugging was complete, correct kernel-based execution would be automatic. That has not been achieved, in part because of the single threaded application environment we set up, and because of other minor differences. Nevertheless, the ability to operate outside the kernel that was made so easy by the stackable architecture and exposure of vnode services, markedly shortened development and testing time.

Performance notes

Ficus is in use at UCLA for normal operation. Its perceived performance is good, but an extensive evaluation is still under way. The major potential performance costs that are observed result from two considerations: execution overhead from crossing multiple formal layer boundaries that might not be present in a more monolithic structure, and additional I/Os from maintenance of needed attribute information. The actual cost of crossing a layer boundary is low – one additional procedure call, one pointer indirection, and storage for another vnode block. In the current implementation, the increased I/O cost can be noticeable, however.

The Ficus physical layer design and implementation accrues additional I/O overhead when opening a file in a non-recently accessed directory. Four I/Os beyond the normal UNIX overhead occur: an inode and data page for the underlying UNIX directory and an auxiliary replication data file must be loaded from disk, as well as the Ficus directory inode and data page. (The last two correspond to normal UNIX overhead.) Opening a recently accessed file or directory involves no overhead not already incurred by the normal UNIX file system.

Conclusions

Our experience with the approach described in this paper has been quite positive. The modularity provided by stackable layers, as well as the simplicity in design and implementation afforded by the optimistic reconciliation approach has been especially significant.

The stackable architecture appears to work quite well: layers can indeed be transparently inserted between other layers, and even surround other layers. A replication service can be added to a stack of "vnode" layers without modifying existing layers, and yet perform well.

The vnode interface is not ideal; a more extensible interface is desired. An inode level interface to files and extensible directory entries would allow us to avoid implementing Ficus directories on top of the UNIX directory service; extensible inodes would allow us to dispense with auxiliary files to store replication data. With these changes virtually all additional I/O overhead over standard UNIX and NFS would be eliminated.

The availability of a general reconciliation service was also very useful. Usually, one must deal with the many boundary and error conditions that occur in a distributed program with a considerable variety of cleanup and management code throughout the system software. Instead, in Ficus failures may occur more freely without as much special handling to ensure the integrity and consistency of the data structures environment. Reconciliation service cleans up later. For example, volume grafting was made considerably easier by the (easy) transformation of its necessarily replicated data structures into Ficus directory entries. No special code was needed to maintain their consistency.

In sum, we are optimistic that services such as those provided by Ficus will be of substantial utility generally, and easy to include as a third-party contribution to a user's system.

References

[1] James E. Allchin. A suite of robust algorithms for maintaining replicated data using weak consistency conditions. In *Proceedings of the Third IEEE Symposium on Reliability in Distributed Software and Database Systems*, October 1983.

[2] P.A. Alsberg and J.D. Day. A principle for resilient sharing of distributed resources. In *Proceedings of the Second International Conference on Software Engineering*, pages 562-570, October 1976.

[3] Joshua J. Bloch, Dean Daniels, and Alfred Z. Spector. Weighted voting for directories: A comprehensive study. Technical Report CMU-CS-84-114, Carnegie-Mellon University, Pittsburgh, PA, 1984.

[4] Michael J. Fischer and Alan Michael. Sacrificing serializability to attain high availability of data in an unreliable network. In *Proceedings of the ACM Symposium on Principles of Database Systems*, March 1982.

[5] Rick Floyd. Directory reference patterns in a UNIX environment. Technical Report TR-179, University of Rochester, August 1986.

[6] Rick Floyd. Short-term file reference patterns in a UNIX environment. Technical Report TR-177, University of Rochester, March 1986.

[7] D. K. Gifford. Weighted voting for replicated

data. In *Proceedings of the Seventh Symposium on Operating Systems Principles.* ACM, December 1979.

[8] Richard G. Guy, John S. Heidemann, Wai Mak, Thomas W. Page, Jr., Gerald J. Popek, and Dieter Rothmeier. *Ficus: A Very Large Scale Reliable Distributed File System. Ph.D. dissertation, University of California, Los Angeles, 1990.* In preparation.

[9] Richard G. Guy and Gerald J. Popek. Reconciling partially replicated name spaces. Technical Report CSD-900010, University of California, Los Angeles, April 1990. Submitted concurrently for publication.

[10] Maurice Herlihy. A quorum-consensus replication method for abstract data types. *ACM Transactions on Computer Systems*, 4(1):32-53, February 1986.

[11] John Howard, Michael Kazar, Sherri Menees, David Nichols, M. Satyanarayanan, Robert Sidebotham, and Michael West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51-81, February 1988.

[12] S. R. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. In *USENIX Conference Proceedings*, pages 238-247, Atlanta, GA, Summer 1986. USENIX.

[13] Thomas W. Page, Jr., Gerald J. Popek, Richard G. Guy, and John S. Heidemann. The Ficus distributed file system: Replication via stackable layers. Technical Report CSD-900009, University of California, Los Angeles, April 1990. Submitted concurrently for publication.

[14] D. Stott Parker, Jr., Gerald Popek, Gerard Rudisin, Allen Stoughton, Bruce J. Walker, Evelyn Walton, Johanna M. Chow, David Edwards, Stephen Kiser, and Charles Kline. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering*, 9(3):240-247, May 1983.

[15] Gerald J. Popek and Bruce J. Walker. *The LOCUS Distributed System Architecture.* Computer Science Series, The MIT Press, 1985.

[16] D. M. Ritchie. A stream input-output system. *AT&T Bell Laboratories Technical Journal*, 63(8):1897-1910, October 1984.

[17] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the Sun Network File System. In *USENIX Conference Proceedings*, pages 119-130. USENIX, June 1985.

[18] Sunil Sarin, Richard Floyd, and Nilkanth Phadnis. A flexible algorithm for replicated directory management. In *Proceedings of the Ninth International Conference on Distributed Computing Systems*, pages 456-464. IEEE, June 1989.

[19] M. Satyanarayanan et al. The ITC distributed file system: Principles and design. *Operating System Review*, 19(5):35-50, 1 December 1985.

[20] Alex Siegel, Kenneth Birman, and Keith Marzullo. Deceit: A flexible distributed file system. Technical Report TR 89-1042, Cornell University, November 1989.

[21] R. H. Thomas. A solution to the concurrency control problem for multiple copy databases. In *Proceedings of the 16th IEEE Computer Society International Conference*. IEEE, Spring 1978.

[22] Gene T. J. Wuu and Arthur J. Bernstein. Efficient solutions to the replicated log and dictionary problems. In *ACM Symposium on Principles of Distributed Computing*, August 1984.

Richard Guy received a B.S. in Math/Computing from Loma Linda University in 1981, and a M.S. in Computer Science from UCLA in 1987. He was a member of the LOCUS project from 1981-87. He has been researching data replication since 1985, and is architect of the Ficus file system. He expects to receive a Ph.D. in Computer Science from UCLA in 1990. Contact him electronically at guy@cs.ucla.edu.

John Heidemann received his B.S. in Computer Science at the University of Nebraska-Lincoln. While there he worked on the MIPS image processing system. Currently he is pursuing a Ph.D. in distributed systems at UCLA. His areas of interest are distributed systems, operating systems, and file systems. He is a member of ACM.

Wai Mak is a programmer analyst at the University of California at Los Angeles. She received her M.S. in Computer Engineering from the University of Southern California in 1989. She is a member of ACM & IEEE Computer Society. Her interest is in computer networks and distributed systems.

Tom Page received the Ph.D. and M.S degrees from UCLA in 1989 and 1983 respectively, with a major field in Distributed Operating Systems. He received a B.S. in Mathematics and Computer Science from Duke University in 1981. While a graduate student, Tom worked on the LOCUS distributed operating system and the Tangram object-oriented modeling environment. He is currently employed as a researcher at UCLA pursuing interests in distributed operating systems and distributed databases.

Gerald Popek has been a professor of Computer Science at UCLA since 1973, having received his doctorate from Harvard. He is best known for his work on the Secure Unix kernel and the design of the Locus distributed Unix system and has authored more than 50 technical papers. Popek is one of the founders of Locus Computing Corporation and is active in various industry service groups including the National Defense Service Board, the World Bank, OSF, and the Unix International Work Group on Multiprocessing.

Dieter Rothmeier is a staff member at the UCLA Computer Science Department. He received a B.S. in Mathematics/Computer Science and an M.S. in Computer Science, both from UCLA. Among his current interests are operating systems and distributed file systems.

USENIX Summer Conference
June 11-15, 1990
Anaheim, California

Binary Emulation of Unix using the V Kernel

David R. Cheriton, Gregory R. Whitehead, Edward W. Sznyter – Stanford University

ABSTRACT

Binary emulation support is essential in future operating systems to support the enormous body of existing application software. Future systems should be able to emulate existing systems, such as Unix, with comparable performance to native execution.

This paper describes the binary emulation of Unix under V, focusing on the kernel and process-level emulation mechanisms. Binary emulation support requires minimal mechanism in the kernel and makes good use of V's server facilities. Binary emulation performance is comparable to that of the native Unix system. For most Unix applications, incompatibilities with the native Unix system appear comparable to those between different versions and releases of the Unix system itself.

1. Introduction

Binary emulation is the execution in one system of the (unmodified) binary file for a program as compiled for a different system. Our particular interest is the execution of Unix.[1] binaries under V [Cheriton 88a]

The ability to execute existing application software in a new operating system is essential for that new system's acceptance, given the enormous investment existing applications represent. Conversely, a new operating system cannot develop the application base to compete with an entrenched system such as Unix without leveraging off the existing application base of such an existing system.

Binary emulation is preferable to other "weaker" forms of emulation that require recompilation, relinking or other modifications to the program for several reasons. First, the source and object forms of the program may not be available to the developers and users of the new system. Second, binary emulation allows a site to maintain one set of binaries for applications rather than two (assuming the site is also continuing to run the original system), saving on disk space, management overhead and possibly licensing costs. Finally, relinking or recompiling may rearrange the binary code and data in memory, thus exposing latent bugs in the program.[2] In particular, if the application executes in exactly the same way within its address space, it is more likely to execute as under the original system, even though the (extended) instructions provided by the operating system are implemented differently. Moreover, our experience indicates that binary emulation does not incur a significant cost in performance or software complexity over the other forms of emulation.

This paper describes the binary emulation of Unix under V. The V distributed system is based on a small distributed kernel that provides basic process management, interprocess communication and virtual memory management (and nothing else). The kernel services are thus limited to the basic operations associated with the process, communication and memory services. They are lower level in nature than conventional operating systems such as Unix. Other operating system services such as file service, scheduling, internetwork services and printing are implemented at the process level (outside the kernel) by server modules. A key goal of the V kernel research has been to demonstrate that a system can be implemented with a small kernel,[3] leading to better reliability, security, maintainability, extensibility and general modularity without loss of performance. The other key focus of V, providing a network-transparent interface to programs running on a cluster of machines connected by a network, is not central to this paper. However, the distributed network-transparent aspect of the V kernel in conjunction with the Unix emulation provides the functionality of a distributed Unix system.

The next section describes the V mechanisms for the emulation of Unix, including the kernel, emulator and system service modules. Although the

[1] Unix is a trademark of AT&T Information Systems.

[2] It is unrealistic to assume that any real program is without bugs, and memory corruption bugs can be harmless until a program is modified or rearranged in memory for some reason.

[3] We use the term kernel here to designate that code and data that are executed and accessed in privileged mode. Only a hardware failure or kernel bug can cause a kernel failure, not any action by an application.

discussion is focused on Unix emulation, these mechanisms appear applicable to the emulation of other systems as well. Section 3 describes the current status of the work and outlines our experience to date. Section 4 provides an evaluation of the performance of the Unix emulator, substantiating our claim that binary emulation does not introduce a significant performance penalty. We close with a comparison to other work in this area and a discussion of our conclusions and future directions.

In this paper, *kernel* refers to the V kernel unless otherwise noted and *application* refers to the program being run in emulation.

2. Unix Emulation Mechanism

Each Unix program executes in its own address space, with references outside its address space going through the system call trap and exception mechanisms. The V emulation mechanism supports the creation of a Unix-like address space for a program, and a mechanism for catching its traps and exceptions, and then handling them as they would be handled by Unix.

The emulation support is divided into three parts: per-program resident emulation code, kernel code, and separate server modules. The resident emulation code is mapped into the *emulator segment* of each program under emulation. The kernel keeps track of the range of this emulator segment, and traps and exceptions incurred while executing outside of this area (i.e., in the original binary) are dispatched to code in the emulator segment instead of being handled by the kernel. Traps and exceptions incurred by code running in the emulator segment are handled directly by the kernel, and thus the emulator segment serves as the interface to the V system. In particular, servers supporting the Unix emulation, such as the file server and the Internet server, are accessible to the code running in the emulator segment.

The emulation components are depicted in Figure 1. Note the arrows that depict the flow of control during the handling of a trap or exception. In particular, note that the resident emulation code jumps directly back to the Unix program without going through the kernel.

The following subsections describe these mechanisms in greater detail.

Kernel Emulator Support

The V kernel support for emulation includes the virtual memory system, an emulator binding mechanism, and the process and interprocess communication facility.

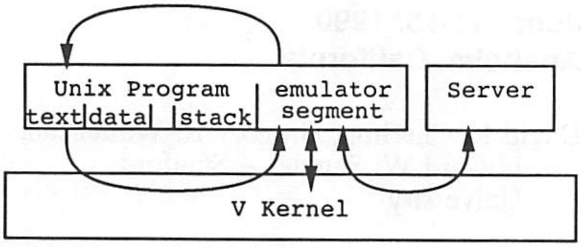


Figure 1: V Emulation Components

Virtual Memory System

The V virtual memory system provides facilities to create a Unix-like address space for each Unix program running under emulation. In particular, the Unix text, data and stack segments are implemented as separate *bound regions* that are mapped to open file objects called *UIOs* [Cheriton 87] corresponding to the program text and backing storage. The data segment is mapped copy-on-write to effect efficient program file sharing and zero-filling of the bss. The bound region mechanism is similar to the mapped file mechanism provided in SunOS 4.0 and later versions.[4]

V's ability to inform arbitrary processes of memory protection violations is used to detect stack underflow and prompt the emulator to increase the stack limit and grow it accordingly. It also allows a program to get control at the point of protection violations and bad address references, similar to the Unix SEGV facility. Mapping a UIO into two separate address spaces supports the shared memory facilities provided in System V (although we have not yet exploited this facility).

The V virtual memory appears to provide basic efficient completeness[5] to emulate any virtual memory functionality that may arise with different versions of Unix. In particular, on many architectures, the V virtual memory system would provide the same mapping and protection with the same granularity as dictated by the hardware. Unix systems would be similarly constrained to match the hardware or a subset of its functionality, facilitating their emulation by V. However, there are some areas of concern.

First, some new hardware architectures do not dictate a specific page size. For example, the VMP multiprocessor [Cheriton 86,Cheriton 88b] uses a virtually addressed per-processor cache that dictates a cache line size but leaves the page size as a decision to the operating system software. Thus, it is possible for a Unix implementation to choose a page size that is incompatible with the V page size.

[4] SunOS is a trademark of Sun Microsystems Inc.

[5] By *efficient completeness*, we mean the ability to implement any ''reasonable'' functionality and interface efficiently.

Another complication arising with these architectures is the ability to control access on the granularity of a cache line, rather than just the page size. For example, on the VMP it is possible to control protection on the granularity of 128 bytes, the cache line size. Similarly, the 801 architecture [Change] provides lock bits on 128-byte units of memory. This fine-grain control is motivated by the requirements of database management systems, particular on-line transaction processing and object-oriented systems. Unix does not currently support this finer grain control at present so the question of emulation support is moot. However, it does complicate the question of what constitutes efficient completeness. We see these issues as warranting further study as such architectures become more common and operating systems adapt to take advantage of these hardware capabilities.

Finally, sophisticated performance monitoring and control functions for virtual memory are an area where it appears most difficult to achieve our ideal of efficient completeness. There is a wide range of possible variables to monitor and it appears expensive to emulate some monitoring facilities without specific kernel support, just relying on the emulator. This is an area of further study.

Emulator Binding

A portion of each process's address space is designated as its *emulator segment*, and when a process executing outside of this area incurs a trap or an exception, control is transferred to the entry point of the emulator segment. If already executing in the emulator segment, the trap or exception is handled directly by the V kernel, as for normal V programs. (Thus, normal V programs execute with an emulator segment that encompasses their entire code segment.)

For the most Unix implementations and architectures, it appears reasonable to place the emulator where the stack would normally start, and simply start the stack at a slightly lower address (assuming the stack grows downwards). The emulator executes on its own stack to avoid overrunning fixed-size user stacks and to further avoid exposing latent application bugs, e.g., programs that depend on values above the stack pointer.

The emulator segment mechanism has several advantages. First, the emulation software has efficient access to the emulated process because the process's memory can be read or written directly from the emulator segment using (unprotected) intra-address space memory accesses.

Second, the overhead to call the emulator and later return to the application is small. In particular, no rescheduling or address space manipulation is necessary. For example, passing control from the kernel trap vector to the emulator, on a DECstation 3100, requires 19 instructions in the V kernel. For programs running in native V mode, the cost of the emulator segment test is roughly 4 instructions on traps and exceptions, making the percentage overhead insignificant for normal V system calls. The emulator can return to the point of the trap or exception using a direct jump, and without invoking a kernel trap.

Finally, the emulator segment approach requires little mechanism in the kernel. Two fields were added to each process descriptor to indicate the beginning and the entry point of the emulator segment, and an additional kernel operation to set these fields, was added. The overall space cost is insignificant.

The alternative of putting the emulation software entirely in a separate address space requires a significantly more expensive invocation and communication mechanism. The performance measurements in Section 4 quantify the benefit of this tight coupling.

The simplicity and generality of the emulator segment approach makes it attractive for emulating operating systems with different user address space layouts as well. For example, it appears feasible to emulate VMS using V running on a VAX (although we have not attempted this emulation to date.)

Several limitations of the emulator segment mechanism are worth noting. First, it assumes that a portion of the address space can be taken for the emulator without affecting the program under emulation. Unfortunately, this violates our ideal of not changing the memory reference pattern of the program at all. Since the amount of virtual address available to the emulated program for stack and heap is reduced, the emulated program may now run out of memory and fail. Also, if the emulator is placed above the stack or below the heap, then a program dependent on a particular starting address for the stack or heap may fail. Alternatively, one could place the emulator segment somewhere between the heap and the stack. Since most current architectures provide only a flat address space, though, choosing a "safe" division point between these two segments is difficult; either the stack or the heap may unexpectedly collide with the emulator. With new 64-bit architectures expected in the future, the stack and heap may well be in separate 32-bit addressed segments, allowing the emulator to reside in its own segment without interfering with the program position or expansion at all.

A second disadvantage is that the program under emulation may access and modify the emulator segment. While the code portion of the emulator can be write-protected, the emulator data cannot be protected from the application on most architectures without an additional protection-changing system call when returning to the emulated code. On an architecture that provides other levels of protection between privileged and user, such as *executive* mode on the VAX, the emulator pages could be set to be writeable only in an intermediate mode. On the VAX, for example, the trap and exception handlers could change the processor state to executive mode before branching to the emulator. The emulator could then switch back to user mode and return directly to the application without going through the kernel. If an intermediate mode is not available, either the emulator would have to run in privileged mode (unacceptable) or the protection of the emulator area would have to be changed on entry and exit (prohibitive).

Current architectural directions are to simplify the protection model of the processor to two states, privileged and user, as found in current RISC processors. However, an additional protection domain for emulation appears relatively easy to provide and worth consideration by architects. It would allow protection of the emulation software without increasing the cost of invoking the emulation software.

V Process and Interprocess Communication Facilities

The V kernel provides efficient interprocess communication facilities, allowing the emulator segment to communicate with other system modules efficiently. The communication uses the (remote) procedure call paradigm, exploiting the V IPC support for efficient request/response or call/return communication. However, the inter-address space call in V is necessarily more expensive than the intra-address space procedure call used inside the Unix kernel. Measurements are planned in the future to determine to the degree that the V IPC introduces a penalty for system performance. If there is a significant penalty, the highly optimized "local" remote procedure calls, as explored in the work of Bershad et al. [Bershad], may be important.

The V kernel supports multiple processes per address space, providing concurrency in the emulator segment similar to that available inside the Unix kernel. For example, the receipt of a signal is handled by a separate process from that executing the application, allowing signals to be handled properly even when the process executing the application is engaged in a system call. Signals make some significant demands on the interprocess communication facility, as described later.

Pipes are the traditional method of interprocess communication in Unix. The V virtual memory system can easily support pipes as part of the file caching and file access mechanisms it already contains, making even high-performance pipe service feasible.

The greater challenge arises with the newer Unix IPC facilities such as System V streams [Ritchie 84].

The System V stream interface is designed around the notion of low-level operations for communicating with, and coupling together Unix kernel-resident modules. From the standpoint of the V kernel, a stream module should execute at the process level in V using emulated stream primitives implemented in terms of the V interprocess communication to communicate with other stream modules and clients. However, this approach appears only feasible for low performance stream applications because of the low-level nature of the stream primitives and aspects of their design that assume a kernel-level implementation. From the standpoint of the V design philosophy, the streams facility is an unfortunate direction for operating systems in providing a lower-level communication interface for applications while not really providing the capability necessary to move modules out of the protected and critical portions of the system, the Unix kernel in this case.

The process management and interprocess communication facilities must deal with the similar issues of efficient completeness as discussed earlier for the virtual memory system. Perfect emulation for process management requires providing the same scheduling behavior, the same virtual timers, real timers and so on. To date, we have not concerned ourselves with completeness in this area, but do see it as an important future effort.

Emulator Segment Software

The emulator segment software receives control of the application when the application executes a trap or incurs an exception. The emulator segment software simply performs the actions the program would expect from Unix.

The emulator also registers as a server in V, allowing it to receive messages from other processes, as required for the implementation of inter-program signals.

A key challenge for the emulator software is to perform with efficiency comparable to that of the Unix kernel, recognizing that it must cross address space protection boundaries to access the services implemented by the V kernel and other server modules while the Unix kernel has direct procedure call access to all (Unix kernel) services.

The emulator segment software is precluded from implementing all the Unix kernel system calls directly (without invoking other V system services) by several considerations. First, some system calls involve access to hardware resources such as disks, networks and memory that are managed by the V kernel, so the emulator software must use the services of the V kernel to implement these calls. For example, **sbrk** involves modification of the address space, which is maintained by the kernel, requiring the emulator to invoke kernel services to implement this Unix system call.

Second, some system calls involve access to data that is shared between different Unix programs. For example, a pipe between two programs needs to transfer data and control between these programs. Thus, either all executing copies of the emulator must share a common data segment for this shared data or else it must be placed in a separate server module and accessed by remote calls from each emulator.

Finally, some system calls involve access to data that requires protection from programs beyond that provided by the emulator segment. For example, the integrity of file protection depends on a trusted and protected component interposed between the application and the files. As mentioned above, in most current architectures, it is expensive to protect the emulator data from modification by the application. Also, an application may be run with a non-standard version of the emulator software even if it is otherwise protected. It is difficult to ensure, and restrictive to require, that every application be run with a trusted version of the emulation software.

To date, we have used separate server modules to handle shared data and protection issues because it appeared simpler to implement in the initial version. Depending on performance experience and further protection capabilities for the emulation segment, future work may investigate use of shared data among the emulator segments of different programs. We note that some of the most demanding shared data portions of the Unix kernel, such as the shared file pointers, are not fully supported between programs in distributed forms of Unix because of the excessive cost and limited use of these semantics. The efforts to distribute Unix are in general reducing the dependence on the semantics that are most difficult to emulate in a modular (and distributed system) such as V.

The following subsections briefly describe the emulation of different classes of system calls with a focus on the key issues and techniques used to achieve efficient execution.

Program Execution and Management

The program execution and management operations in Unix include **fork**, **vfork**, **exec** and **wait** with its variations.

To implement **fork**, the emulator creates a new address space, copies the current address space to the new address space (duplicating virtual memory bindings instead of actual copying where possible), creates a new process in the new address space, copies the current process state to the new process, and starts the process. The new process initializes itself and returns from the **fork** call as the child process. The **vfork** handler creates a child process in the current address space, and only builds a new address space if **exec** is called, then using V's process migration to move the already created child process to the new address space.

Because the emulator segment is preserved across the **exec** operation, it provides a code and data segment in which execution can proceed while the rest of the address space is "refurbished," the same as the role the Unix kernel plays in performing an **exec**. The **exec** mechanism simply replaces the application code and data with the new program. Open files and other appropriate data structures are naturally inherited because of the retention of the emulator state.

When an emulator receives an exit indication from the executing program. it uses the V IPC to inform its parent before it dies. The various wait system calls need only wait for this message. The emulator does not explicitly support a zombie state as provided in Unix but ensures that exit status is delivered to the parent.

It is possible to destroy an emulated process using native V mechanisms; the process will disappear without returning an exit status. The emulator maps this case onto the process being killed without a core image. An application cannot produce this type of anomalous behavior for itself or as a result of actions by other emulated Unix applications.

The creation of a new address space is handled entirely by the emulator in conjunction with the V kernel, but the existence of the new address space must be recorded in a system-wide "database." This database is maintained by a server module, called the team server, which supports queries about executing programs and performs longer-term scheduling.

I/O

The emulator implements the Unix file descriptor data structures, mapping I/O operations onto V UIOs. The UIO interface supports reading and writing of blocks, but it does not maintain current file position or other client-specific information; it is close enough to the basic block device interface that this mapping is straight-forward. The UIO interface is similar to the VFS interface [Sandberg] except the UIO interface is available outside the kernel and uses a stateful model of access to open file objects rather than a so-called state-less interface to the actual files.

The performance of the emulator benefits from several aspects of Unix I/O. File I/O is the most intensive activity in many Unix systems. Most I/O that is buffered or cached in Unix, such as ordinary file I/O, can also be cached in V. UIOs can either be accessed directly or using a cached access provided by the V kernel virtual memory system. For standard files, the emulator accesses files using the caching facility, mapping multi-block portions of the file into its address space. Read and write operations can then handled by copying data to and from the application's buffers and the emulator's data segment, with additional remapping of these buffers as required. For example, the emulator might map 16 kilobytes of a file at a time, so a kernel operation (to rebind the buffer to another part of the file) is only required every 16 read operations if the application is reading sequentially 1 kilobyte at a time. Since the emulator segment and the application are in the same address space, buffered data is moved by a simple copy, instead of between the privileged kernel data and user space as in Unix. This is more efficient on many architectures.

Direct types of I/O are largely limited by the performance of devices, rendering the cost of the emulation insignificant. For example, terminal I/O generally operates in the kilobit per second range, which is easily handled by the emulation.

The limited number of open files allowed in standard Unix makes it relatively easy for the emulator to implement this mapped access in the bounded space of the emulator segment. Future versions of Unix will remove the limitation on the number of open files but future architectures will likely provide a segment memory with 64-bit addressing, eliminating the current space constraints on the emulator.

Sockets are implemented in a similar fashion to the file support, with the emulator translating between standard socket operations and V IPC interaction with the V Internet server. When emulation runs on 10 MIPS workstations such as the DECstation 3100 connected by 10 Mb Ethernet, the processor performance relative to Ethernet performance is such that emulator overhead is not significant. However, as network performance moves to the 100 Mb range and more, further optimizations will be required. Currently, there is an extra data copy as a result of using an Internet server outside of the kernel. However, it appears feasible to eliminate this copy with further optimizations in the IPC mechanism, and with this optimization, it appears that the protocol processing overhead would render the overhead of emulation (an extra context switch and kernel call) insignificant.

The final aspect of implementing the I/O services is supporting signals, as described below.

Signals

The signal support includes the calls to send signals, set and clear the signal handlers, and the **select** and **pause** operations to receive signals.

A signal is sent to a program by sending a V IPC message to the emulator server interface, which is waiting to receive messages. The emulator maintains the signal state of the program and then mimics the appropriate Unix action. For example, if the signal is blocked, the signal is recorded but takes no action until the process unmasks the signal, at which point it invokes the signal handler. Otherwise, the emulator's server must ensure that the emulated process calls the handler routine in a timely manner. This is complicated by the fact that process scheduling is being handled by the V kernel, not the emulator, and involves a V kernel call to either cancel a blocked operation in a system call handler, thus aborting the system call, or force a jump into the emulator.

Some signal events, such as memory access and protection violations and instruction and arithmetic exceptions, are caught by the V kernel. Normally, a message would be sent to a server process that would then start a debugger on the program; the emulator server receives these events by registering as the exception handler for the application process, so it receives the message instead.

Other signals correspond to events in the system that are not naturally tied to the application process, particularly given that V minimizes the connection-like state between services and their clients. For example, an open file in V may be accessed by any number of clients; the server does not keep track of the clients, and allows access limited only by protection restrictions. Moreover, in the V model, clients request information from servers, rather than servers volunteering the information. In particular, there is no notion of server notification of clients in this design unless a client has a call outstanding to the server.

To minimize the modifications to the V servers, our initial implementation of signals uses small helper processes in the emulator that query the servers for signal events. If a signal event occurs, the server responds to the helper process which then reports back to the emulator. For example, with select, there is a helper process created for each open file on which the select is performed, to query all the servers associated with all the selected open files.

This approach has several disadvantages. First, the approach does not appear to generalize easily to handle process groups. Second, there is a per-client system cost associated with creating and maintaining these helper processes. Finally, helper processes impose a load on the servers, both at the level of remembering and responding to them as well as at the transport level (if the server is across a network) by keep-alive retransmissions while the helper is waiting for a response.

The alternative we are planning to explore entails adding a notification facility to the servers so that they send to the interested parties using the V IPC. In this approach, the server records the V process identifiers associated with programs interested in particular events. When an event of interest occurs, the server sends a notification to the process or processes associated with the identifier. To handle Unix process groups, V process groups are used to represent multiple processes; the server records the identifier for the process group. Each emulator joins the V process groups corresponding to the Unix process groups to which the application process belongs. The mechanism for sending notifications to a single process or a group of processes is identical in V, thus the system calls **kill** and **killpg** use the same code.

Using this approach, the number of messages a server sends corresponds to the number of distinct process groups interested in the event, not the total number of processes in all the groups, as with the current approach. Also, a context switch for activating the helper processes is eliminated. Finally, the need for the helper processes is reduced, reducing system overhead per application and the time to communicate a signal.

We expect to report on our experience with this alternative at a later date.

Memory Management

The **sbrk** system is implemented by expanding or contracting the address space bindings using the V virtual memory system primitives. Automatic stack extension is implemented similarly, except it is invoked by a memory exception caught by the emulator. Other memory management calls have not been implemented at this stage.

Starting an Emulated Process

Unix emulation is initiated under V using the V library routine **UnixemExec**, which starts a Unix program in execution under the Unix emulation software, using the arguments, environment and I/O specified in the parameters to the routine. At this time, we have used this routine to start an instance of the shell, allowing additional Unix applications to be executed using the facilities of the shell. One could also use this mechanism to start more basic Unix software, such as **getty**, to give the impression at login of a Unix system. **UnixemExec** can also be used to execute individual Unix programs from a V program to, for example, access functionality not available directly under V. The latter is facilitated by **UnixemExec** using the same procedural interface as the routine for executing standard V programs under V.

UnixemExec creates an address space and initial process, loads the resident Unix emulation code, argument list, and environment strings into the emulator segment at the top of the address space, and then starts the process running at an initialization routine in the resident emulation code. Except for the load address, this procedure is the same as that for starting a normal V executable. The resident emulation code initializes itself, hooks itself into the kernel by registering its start address and entry point, and then passes control to the Unix **exec** system call handler with the appropriate arguments. This causes the specified Unix program to be loaded into the Unix address space and executed.

Support for Different System Versions

Different versions of Unix can be supported under V simultaneously by executing applications with different versions of the emulator or by passing version information to the emulator at run-time. To date, we have executed standard BSD and Ultrix binaries. The differences between the two are handled by conditional compilation in the source code of the emulation software. This support for different versions of Unix points out the benefit of allowing different Unix applications in the system to run with different versions of the emulator. In particular, the system can be extended to handle new versions of Unix without modifying the existing emulation code, benefiting the size and stability of the emulator. It also makes it feasible to restrict the set of system calls available to a program, for example, to test the portability of a program or for security reasons, by using a restricted version of the emulator. Similarly, it may be useful to have a highly instrumented version of the emulator to aid in the debugging of applications.

System Server Modules

The three key V system servers roughly correspond to three large pieces of the Unix kernel, namely the file system, network services and scheduling. These modules were not originally designed to support Unix emulation. In fact, they were developed primarily to demonstrate the functionality of the V kernel with little concern for Unix compatibility or efficient completeness. In this vein, there are numerous incompatibilities at this time, some of which we have masked and others we have not.

The Internet server has the least difficulty with compatibility because the key aspects of the service visible to the user are the socket interface and the machines and services accessible using the Internet protocols. The former is relatively easy to emulate whereas the latter is the same, independent of the emulation, because it is a property of what is connected to the Internet.

The scheduler has received relatively little attention to date, especially regarding efficient completeness. A few factors complicate the situation. First, Unix maintains information such as the number of voluntary versus involuntary context switches, and V does not at this time. While V could be modified to maintain this information, such a move has the flavor of ''adding Unix into V,'' rather than emulating Unix per se.

Second, it is possible for Unix programs to be sensitive to specific scheduling algorithms used in Unix. Even if V happened to be running the same algorithms, the scheduling decisions would be based on all running programs, not just the emulated programs visible to the application. Thus, the behavior could be appear to be different than Unix.

Finally, many Unix programs access information about running programs in ad hoc, undocumented ways, a familiar example being the reading of **/dev/kmem**. This level of compatibility seems infeasible to support, and seems unimportant to most Unix programs of interest.

The file server and V file system have the same problem as the scheduler, only worse. Unix programs expect a particular file name hierarchy for standard files, so emulating Unix file semantics is not sufficient. For some programs, there must be the same database of files and directories in the file server, particularly standard ''system'' files. Moreover, actions taken against these files are often expected to affect the system behavior, an example being ''lock'' files. This collection of system files represents another unstructured and undocumented aspect of Unix that is difficult to emulate.

3. Experience

The V system at Stanford currently runs on a cluster of VAXstation II, DECstation 3100, and Sun 3 workstations sharing an Ethernet. Also on the Ethernet, and providing file service, are a VAXstation II running Unix BSD 4.3, and a VAX 8350 and a DECstation 3100 running Ultrix. Binary emulation is currently supported in the VAXstation II and DECstation 3100 kernels.

The emulator is comprised of about 200 lines of assembly code and about 8000 lines of C code (including comments). The executable size (text and data) is 87 kilobytes for the VAXstation II and 148 kilobytes for the DECstation 3100. Currently implemented are 79 of the 110 BSD 4.3 system calls and 3 of the vendor specific Ultrix 2.0 system calls. These include the calls for process and memory management, signal management and delivery, time and host queries, descriptor management and I/O, filesystem management, and internet domain sockets. The calls not yet implemented are for protection, resource control, process groups, device management, messages, and semaphores. Many of these are ''a small matter of programming.''

Even though the emulator does not yet emulate the complete functionality of Unix, it does support a large number of interesting binaries: **/bin/sh**, **/bin/csh**, **/bin/cc**, **/bin/make**, **/usr/bin/nroff**, **/usr/ucb/vi**, **/usr/ucb/finger**, etc. This is not completely accidental, as the requirements of various interesting programs have influenced the decisions on what functionality to emulate next. Also, it has become clear that it is not economic to implement complete emulation. At some point the cost of adding emulation of additional functionality will outweigh its benefit. As mentioned previously, **/dev/kmem** seems like a good example of this point.

An unexpected complication arose with the MicroVAX II in the course of this work. The MicroVAX II hardware implements a subset of the full VAX instruction set and the V kernel had not bothered to emulate the remaining instructions. We discovered that the Vax Unix binaries used some of these unimplemented instructions, causing the programs to fail. One could argue that the V kernel should emulate these missing VAX instructions to support the full VAX architecture. However, it is a testimony to the emulator software that we were able to easily add emulation for these instructions to the Unix emulator and allow the binaries to execute, without modifying the V kernel.

4. Performance

A preliminary performance evaluation of the emulator was carried out to determine the emulation performance relative to running under native Unix. We measured the basic cost under emulation for a simple system call, I/O performance, and the execution times for a few applications. The measurements were done on a DECstation 3100 running/emulating Ultrix, and on a VAXstation II running/emulating BSD 4.3. Overall, the results are encouraging.

Simple System Call

The overhead for a system call (trapping into the kernel, passing control to the emulator, and returning control to the process) is comparable to (and in fact less than) that for Unix (trapping into, and returning back from, the kernel). This is shown by the times for the **getpid** system call in the following table.

The time is comparable for the emulator because the emulator executes in the same address space and protection domain as the application. The cost of crossing back and forth across the protection boundary, which is paid in both the emulator and Unix, is the dominant cost.

I/O Performance

The I/O performance for cached files was measured to quantify and validate the expected benefits from caching and the intra-address space copy for delivering the data. The measurement used was the time to read 16 kilobyte blocks contained in the local file cache, and as is shown by the **cacheread** times in the following table. Note that times are given for both page aligned and unaligned read buffers. V is able to take advantage of page aligned buffers, using mapped I/O, for a significant performance improvement (almost 20 in this case). Comparing the Ultrix and V times, for unaligned buffers, it is seen that V is roughly 22 faster at reading a 16 kilobyte block from the file cache. We conjecture that this performance benefit arises from the use of an (unprotected) intra-address space copy to deliver the data from the emulator segment in V versus the protected copy required to copy the data from the Unix kernel.

The I/O performance for reading from a fileserver was measured to examine the penalty of using user-level V services in place of similar services provided by the Unix kernel. The measurement used was the time to read 1 kilobyte blocks from the file block cache on a remote fileserver. The fileserver was a DEC VAXstation II. As is shown by the **serverread** times in the following table, there is roughly 0.95 ms of overhead, for a 14 performance penalty, in this scenario. This overhead represents the scheduling and communication costs of placing the fileserver outside of the kernel. Note that this overhead would have appeared to be even less significant if the file blocks had been read from disk because of the added disk latency swamping the overhead of emulation. This overhead is avoided completely in the case that there is a local cache hit, so the overall performance penalty should not be significant for average workloads on machines with sufficient memory.

DECstation 3100		
Test	Ultrix	Emulator
getpid	0.028 ms	0.021 ms
cacheread (aligned)	3.520 ms	2.200 ms
cacheread (unaligned)	3.520 ms	2.740 ms
serverread	6.593 ms	7.540 ms

Applications Performance

Several applications programs were run to verify the performance of the system under real use. The VAXstation II was used for this purpose since that version of the emulator has been around longer and has a more complete runtime environment available. It should be noted that the performance of the DECstation 3100 version of the emulator has been slightly better, relative to Unix, than the VAXstation II version. We believe that this is mostly attributable to the reduced procedure call overhead on the R2000 architecture. Thus, when the environment becomes available, the performance of applications programs on the DECstation 3100, relative to Unix, should be as good or better than on the VAXstation II.

The times quoted in the following table are elapsed real time for lightly loaded systems. Thus, they can be taken as an estimation of the system's response. The first time is for formatting the **/bin/csh** manual page. Note that the times are comparable. This demonstrates that, with an average workload, V's high performance I/O can compensate for the overhead of providing traditional Unix kernel services outside of the kernel.

The second time is meant to measure the cost of the **fork** and **exec** system calls. It is a **/bin/sh** script that loops running **/bin/echo**. The cost of a **fork/exec** can be seen to be roughly 770 ms higher for the emulator than for BSD 4.3. This extra cost is primarily due to the number of operations required to implement this functionality in terms V kernel primitives; the V kernel does not provide a **fork** primitive.

The third time is for **/usr/ucb/finger** to list the users on one of our local machines. In the emulator, this uses the user-level V internet server. Note that the times are comparable.

VAXstation II		
Test	BSD 4.3	Emulator
nroff -man csh.1	129 s	123 s
sh exec/fork	0.13 s	0.90 s
finger @pescadero	2 s	2 s

In summary, our measurements on this preliminary version of the emulator suggest that conventional Unix programs should execute as fast, if not faster, when run under the emulator than when run under native Unix on the same architecture. We use the term ''conventional'' to rule out programs that make extensive use of new Unix facilities such as streams, messages and semaphores which we have yet to implement, and with which we have some concerns.

5. Related Work

The Mach project [Rashid] is using some similar techniques to those employed for emulation in V.[6] However, the Mach project is effectively working from a version of Unix and attempting to remove functionality from the Unix kernel. Thus, Mach must deal more with preserving compatibility than achieving it in the first place. In particular, functionality is moved out of the Mach version of the Unix kernel if it is feasible to do so. In contrast, we are starting with a largely incompatible, independently developed operating system kernel and attempting to achieve binary compatible under emulation with minimal changes to this kernel.

Chorus [Rozier] is an example of another system providing Unix binary compatibility. However, as we understand it, the code providing the Unix kernel interface is linked with the Chorus kernel to provide this emulation support. Thus, they do not preserve the small kernel protection boundary as we

[6] In particular, the design of the emulator segment was influenced by discussions with Rick Rashid and his work on the Mach operating system.

have done in V. Topaz [McJones] is another example of a system providing a high degree of binary Unix compatibility. The Topaz emulation is centralized in a single operating system module, rather than being distributed, as in the V implementation. The centralized implementation makes interprogram services easier and more efficient to implement at the cost (we expect) of increasing the cost of system traps handled directly by the emulator segment in V. Topaz also allows programs using Unix emulation to directly access native Topaz services whereas V currently strictly prohibits Unix programs from directly accessing V native services. It seems appropriate to provide both options, to accommodate evolving programs to effectively use the new native system in the former case, and to ensure original system conformance in the latter. Other systems include KSOS [Walker], which is Unix layered on top of a secure kernel, and Cray's COS and Unicos [Ritchie 88], which provide a virtual-machine guest facility in which Unicos can be run. There are also what are effectively reimplementations of the Unix kernel, such as Sprite [Ousterhout]. While Unix binary compatible, Sprite does not face the same emulation problems experienced in V because the Sprite kernel functionality is basically defined by Unix.

Another category of emulation projects are those providing IBM PC and MS-DOS binary compatibility [Nash,Sarno], in some cases on non-x86 architectures. For example, Sun Microsystem's *DosWindows* application provides binary emulation of both MS-DOS and the Intel 286 processor, running on a SPARC workstation. These emulations differ in dealing with a single-address-space operating system in which a key issue becomes correctly emulating access to devices. In addition, the emulations such as *DosWindows* clearly differ in emulating an entire processor instruction set, not just operating system calls, leading to a performance penalty over native execution. Due to their proprietary nature, we do not know whether these emulations use the same techniques as we have employed. However, we expect that an emulation of both operating system and processor, such as generally required for IBM PC emulation could be implemented using the facilities and techniques we have described.

Another emulation approach is to provide a compatibility library against which programs can be relinked. This approach was used previously in V, but given our experience with binary compatibility, we see it having significant disadvantages over binary compatibility and no real benefits. In particular, the amount of software required in a V implementation for binary-compatible emulation over source-level emulation is insignificant and, as our performance measurements show, performance does not suffer.

In general, we see our work and the cited work as evidence that emulation has been recognized as an important operating system facility. Our work is distinguished by starting with an advanced research operating system developed with little concern for compatibility, and retrofitting the system with support for emulation without compromising the original design goals.

6. Conclusions

Binary Unix emulation can be provided in V for a large and useful subset of Unix applications with a relatively small amount of software, resulting in performance comparable to that of a native Unix system. Moreover, the emulation has been implemented using mechanisms that are general-purpose and do not compromise the design for Unix idiosyncrasies. In particular, the small size and tight protection aspects of the V kernel have been retained.

Complete compatibility appears to be both impossible and ill-defined in general. There are significant differences between different versions of Unix, and even between different releases of the ''same'' version of Unix. We claim that the level of incompatibility achievable between the Unix emulator and the native system for most Unix applications is comparable to that experienced between successive releases of the Unix system itself. For example, in our goal of letting the Unix application execute the same sequence of memory references in its address space as it would in native mode, the major exception is the repositioning of the process stack to accommodate the emulator segment. However, different releases of Unix also reposition the base of the stack because of growth in the user area, potentially exposing the same application problems. Fortunately, some aspects of Unix that are difficult to accommodate in V, such as shared file pointers, are difficult to support in a distributed environment, even for native Unix, and are thus becoming less used by applications. Interestingly, with the transparent distributed operation supported in V, it is even an issue what constitutes one Unix system in emulation under V. In particular, is it one host, a group of hosts or a subgroup of the programs executing on a cluster of hosts? This question requires further thought, and is being addressed to some degree as Unix becomes increasingly distributed.

There are several important criticisms applicable to this work. First, to some degree we have tackled the easy part of Unix. That is, Unix from its original conception defines a clean, encapsulated interface to the operating system for applications compared to previous operating systems. The application is completely encapsulated in its address space and interfaces with the kernel using relatively simple system calls. In contrast, contemporary operating systems at the time of the original Unix development, such as OS/360, revel in complex control block interfaces and murky protection boundaries between application and system. A weak point of Unix is the *ad hoc* databases used to maintain system information, much of which is as loosely defined as the control blocks of older operating systems. A significant improvement over Unix would provide a means to maintain this information in a unified, well-defined way and emulate the Unix facilities for backward compatibility. In essence, the well-structured portions of the system are easiest to emulate but least important to replace while the least structured are the most difficult and most important to improve.

This criticism identifies a legitimate limitation of our work to date. However, database management systems, including recent object-oriented management systems are providing a mechanism for unifying system information management, so a promising solution is at hand (and we hope to experiment with these mechanisms in the future). Also, the V kernel provides a good set of primitives for efficient implementation of these database facilities both for applications as well as system management. Finally, the dual developments of network management standards and the adaptation of Unix to the distributed environment suggests that Unix and its applications may evolve to use better system management interfaces for reasons independent of this work.

A second related criticism is that we have not shown we can efficiently emulate some of the relatively new primitives added to various versions of Unix, such as the streams facility in System V and the capability-based message facility in Mach. These facilities will be difficult to emulate efficiently without effectively implementing them at the kernel level because they are low-level in nature. We do not see these facilities as a major problem for several reasons.

First, most application writers avoid the use of these new primitives to ensure portability of the application across different versions of Unix. Second, the main users of these new low-level primitives (to date) are system services, such as networking and database management systems, that are appropriate to be modified, rewritten or totally replaced, rather than emulated, to take best advantage of the new system. That is, these system services are properly part of the "new" system and therefore should be provided as an integral component. For example, a rewrite may be motivated in making the transition between a low-level kernel implementation of a system service, as in Unix, to a process-level implementation, as in V. Finally, a modest or low performance implementation of these low-level primitives is possible using the emulation technology described here, allowing a new system to acquire the functionality of these services quickly and then evolve in performance by modification to their implementation.

These low-level primitives that have been added to Unix kernel interface can be viewed as violating its original design goals, which focused in part on providing system primitives that were close to that desired by the applications. For example, the byte-stream I/O of Unix (including the original implementation of **putchar** as **write(1,&c,1)**) was relatively novel at the time Unix was conceived in contrast to the "Rube Goldberg" record I/O of various mainframe operating systems of the time. The original Unix systems calls are relatively straightforward to emulate precisely because they are (well-designed) high-level primitives. In this vein, the System V streams facility seems like an unfortunate addition to Unix by providing a complex interface to applications that is difficult to emulate and that was not designed to facilitate moving function outside of the Unix kernel or making services distributed.

From these observations, we propose that there should be different levels of system interface, and in particular a high-level restricted system call interface corresponding to more or less the original Unix system interface, and specifically excluding the various new low-level primitives. Although many application writers already implicitly work within such an interface to facilitate portability, it would be useful to have this subset well-defined. We understand that some POSIX working groups are beginning to define *application environment profiles* including one for "traditional Unix" that are somewhat of this flavor. Ideally, these profiles should be enforced by the system for programs so designated. We note that the emulator mechanism provides a simple means of enforcing a limited interface or profile to applications, simply by limiting the set of Unix system calls implemented by the version of the emulator provided to the application upon execution. In this sense, the emulator running under V can be viewed as providing and enforcing a reasonable high-level interface to applications to ensure their future portability rather than an evolutionary path for applications to move to using low-level primitives such as the V kernel operations. That is, even new applications should use the basic Unix primitives if at all feasible rather than moving to using the low-level V primitives even if V was to become a standard operating system.

Under this proposal, a restricted subset of programs would use the various new primitives, many of which would be system server modules. Thus, rather than questioning how to emulate (for example) Mach IPC using V IPC, the real issue would be viewing the two facilities as competing sets of primitives for implementing a high-level application interface, namely the Unix system call interface. In effect, we are reiterating a lesson learned previously from work on portability, just now applying it to binary portability: *High-level primitives are more portable than low-level primitives.*

As a final criticism, one might question the degree that V is effectively really just an alternative implementation of Unix, rather than providing an emulation of Unix. This point is valid in several senses. First, as we propose above, applications should not in general use the V primitives, so to an application writer it is just another Unix. Second, V is compatible with Unix in some ways because of related history and because of changes during this emulation work. However, this view fails to recognize fully the contribution of this work. In particular, V is capable of emulating other operating systems as well as Unix, such as VAX/VMS, MS-DOS and so on, using similar mechanisms to that described here.

In closing, based on our experience with emulation in V we see emulation support as a good test of an operating system design, ensuring that the new system incorporates (at least in its interfaces) the lessons the previous systems have learned from hard experience. In this vein, it is useful to examine the design of the system service interfaces from the standpoint of what we have called *efficient completeness.* The inability of a new system to efficiently emulate some aspect of an existing system is likely either a deficiency in the design of the new system, or a really unfortunate aspect of the old system, and the latter should not be too quickly assumed. While V has generally fared well in supporting Unix emulation, this exercise has pointed out some areas for improvement.

The work reported here represents an initial effort. Based on the positive results to date, we plan to improve the fidelity of emulation as well as improve the V kernel mechanisms that support emulation. We expect to report on this work in a future publication.

Acknowledgements

The comments of Paul McJones of the DEC Systems Research Center, Hendrik Goosen of Stanford, Kieran Harty of Stanford, and the referees lead to significant improvements in the paper.

References

[Bershad] Bershad, B. et. al. ''Light-weight Remote Procedure Call.'' *ACM Trans. on Computer Systems*, 8(1):12–46, Feb 1990.

[Change] Change and Mergen ''801 Storage: Architecture and Programming,'' TOCS 6(1), Feb. 88.

[Cheriton 86] Cheriton, D.R., Slavenburg, G., and Boyle, P. ''Software-controlled caches in the VMP multiprocessor.'' In *13th Int. Conf. on Computer Architectures.* ACM SIGARCH, IEEE Computer Society, Jun 1986.

[Cheriton 87] Cheriton, D.R. ''UIO: A uniform I/O interface for distributed systems.'' *ACM Trans. on Computer Systems*, 5(1):12–46, Feb 1987.

[Cheriton 88a] Cheriton, D.R. ''The V Distributed Operating System.'' *Communications of the ACM*, 31(4), Apr 1988.

[Cheriton 88b] Cheriton, D.R., Gupta, A., Boyle, P., and Goosen, H. ''The VMP multiprocessor: Initial experience, refinements and performance evaluation.'' In *Proc. 15th Int. Symp. on Computer Architecture*, May 1988.

[McJones] McJones, P.R., and Swart, G.F. ''Evolving the Unix System Interface to Support Multithreaded Programs.'' *SRC Research Report 21*, DEC Systems Research Center, Sep 1987.

[Nash] Nash, H. ''The design and development of a software emulator.'' In *Proc. 34th IEEE Computer Society Int. Conf. (COMPCON)*, San Francisco, CA, Feb 27–Mar 3 1989.

[Ousterhout] Ousterhout, J.K. et. al. ''The Sprite Network Operating System.'' *IEEE Computer*, 21(2), Feb 1988.

[Rashid] Rashid, R., and Baron, R. ''Mach: A Foundation for Open Systems.'' In *Proc. Second Workshop on Workstation Operating Systems*, Pacific Grove, CA, Sep 27–29 1989.

[Ritchie 84] Ritchie, D.M. ''A Stream Input-Output System.'' *AT&T Bell Laboratories Technical Journal*, Oct 1984.

[Ritchie 88] Ritchie, D.M. ''A Guest Facility for Unicos.'' In *USENIX Proc.–Workshop on Unix and Supercomputers*, Pittsburgh, PA, Sep 26–27 1988.

[Rozier] Rozier, M. et. al. ''Chorus Distributed Operating Systems.'' *Technical Report CS/TR-88-7.8*, Chorus Systems, Feb 1989.

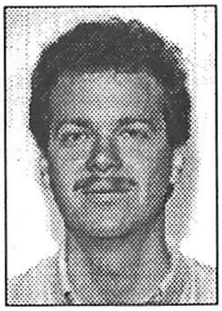
[Sandberg] Sandberg, R. et al. ''Design and Implementation of the Sun Network File System.'' *Usenix Association Conference Proceedings*, Summer 1985.

[Sarno] Sarno, K. ''Two that tango on the 80386.'' *Unix Review*, 6(1), Jan 1988

[Walker] Walker, S.T. ''The Advent of Trusted Computer Operating Systems.'' In *AFIPS Conf. Proc.*, Anaheim, CA, May 19–22 1980.

David Cheriton is an Associate Professor of Computer Science and Electrical Engineering at Stanford University. His research includes the areas of high-performance distributed systems, shared memory multiprocessor architectures and high-speed computer communication. Prof. Cheriton received his Ph.D. in Computer Science from the University of Waterloo in 1978. He subsequently spent 3 years at the University of British Columbia in Canada. For the past 8 years, he has been at Stanford.

Gregory Whitehead just completed an M.S. in Computer Science at Stanford University. He also has a B.S. in Computer Science and Engineering from the University of California at Davis. While at Stanford, he was a Research Assistant in the Distributed Systems Group and implemented the Unix Emulator described in this paper.



Ed Sznyter is a Research Programmer in the Distributed Systems Group at Stanford University, primarily working in the areas of virtual memory architectures and distributed systems. He holds a B.S. in Applied Mathematics from Carnegie Mellon University.

USENIX Summer Conference
June 11-15, 1990
Anaheim, California

Unix as an Application Program

David Golub, Randall Dean, Alessandro Forin, Richard Rashid – School of Computer Science; Carnegie Mellon University

ABSTRACT

Since March of 1989 we have had running at CMU a computing environment in which the functions of a traditional Unix system are cleanly divided into two parts: facilities which manage the hardware resources of a computer system (such as CPU, I/O and memory) and support for higher-level resource abstractions used in the building of application programs, e.g. files and sockets. This paper describes the implementation of Unix as a multithreaded application program running on the Mach kernel. The rationale, design, implementation history and performance of the system is presented.

Introduction

It is both possible and rational to think of Unix (Unix is a trademark of AT&T Bell Laboratories)not as an operating system kernel but as an application program – a server or set of servers that can provide client programs with specific programming abstractions. For the last twelve months we have had running at CMU a computing environment in which the functions of a traditional Unix system are cleanly divided into two parts: facilities which manage the hardware resources of a computer system (such as CPU, I/O and memory) and support for higher-level resource abstractions used in the building of application programs, e.g. files and sockets. The hardware management duties of Unix – virtual memory, scheduling and device management – are provided in an operating system environment independent way by the Mach kernel [1]. The key, recognizable Unix facilities – such as Berkeley files, sockets and ttys – are literally provided by an application program running on top of Mach. Even to the sophisticated Unix systems programmer the result is a seamless environment in which Berkeley 4.3BSD, Ultrix or SunOS 3.x binaries (depending on the machine type used) continue to run. Performance is comparable in a number of respects to that of earlier Mach releases and commercial Unix implementations.

The notion that Unix can be implemented by an application program acting as a server is the logical culmination of a longstanding trend in OS design. Over the last decade system programmers have increasingly relied upon the client/server implementation model as they have expanded the functionality of computer environments. Network file services, name services and database services are just a few of the server-provided facilities which have been added to traditional operating systems. Fueling this trend has been the recognition of the considerable advantages to be gained by the client/server approach, such as modularity and network transparency.

Beyond the obvious advantages common to all client/server systems, treating Unix as an application program has a number of interesting implications:

- tailorability – Versions of Unix such as 4.3BSD, Posix and System V.4 can be treated simply as different applications which can be purchased separately and potentially run simultaneously or even side by side with other OS environments.
- portability – Nearly all of the code which constitutes Unix is independent of a machine's instruction set, architecture and configuration.
- network accessibility – A Unix Server need not reside on the same machine as a Unix client.
- extensibility – New versions of Unix can potentially be implemented or tested alongside existing versions.
- real-time – Traditional barriers to real-time support in Unix can be removed both because the kernel itself does not have to hold interrupt locks for long periods of time to accommodate Unix system services and because the Unix services themselves are preemptable.

Our work does not represent the first time that Unix or another operating system has been implemented as an application program. Other such implementations have, however, frequently started from a rather different notion of the relationship between the system kernel and the supported OS environment. The approach we have taken with Mach was not to implement a virtual machine (as in IBM's CP/67 [4]) or to layer the kernel on a simple message engine (as in AT&T's MERT [6]) or to use a global shared communication area (as in Taos [8]) or to load operating system environment specific emulation-assist code into the kernel (as in Chorus [2]). Rather, we have taken advantage of the fact that Mach provides for the manipulation of system resources through a small set of machine-independent abstractions and for the integration of memory management and communication functions. All functionality pertaining to the implementation of Unix services is performed by a multithreaded Mach task which takes advantage of the Mach IPC, scheduling and virtual memory services. Unix facilities are provided by an application program running on top of Mach and no Unix-specific functionality exists within the kernel.

This paper describes our implementation of 4.3BSD as an application program. In particular, it examines the Unix Server, the key features of Mach on which our implementation depends and some of the challenges we encountered in the development of the system. The functionality and performance of the resulting system is also presented.

The Organization of Unix as an application program

The implementation of Unix binary compatibility on Mach divides cleanly into two parts: a Unix Server which provides the system services and resources commonly associated with a 4.3 BSD Unix environment and a Transparent System Call Emulation Library which operates within the address space of a Unix application. The Mach kernel provides the key support facilities upon which both the Unix Server and the Transparent Emulation Library depend. In particular Mach provides for interprocess communication, memory object creation and data management, scheduling and trap redirection.

Key Mach Features

It is difficult to describe the Unix Server and Emulation Library without briefly describing the most important details of the Mach kernel on which they both depend. A more detailed description on Mach and its abstractions can be found in [1]. The key features of Mach used in the emulation of Unix services are:

- interprocess communication,
- memory object management,
- scheduling,
- system call redirection,
- device support, and
- multiprocessing support.

Interprocess communication

Mach defines an address space and protection domain to be a task and a CPU flow of control to be a thread. Mach provides interprocess communication between threads through constructs called ports. Ports are protected with a capability mechanism and only Mach tasks with appropriate send or receive capabilities can access a port. All services, resources and facilities within the Mach kernel and between Mach tasks are represented as ports. Mach tasks, threads and memory objects are, for example, all manipulated by sending messages to ports which represent them. As such, the Mach port facility can be thought of as an object reference mechanism.

Memory object management

The address space of a Mach task is represented as a collection of mappings from linear addresses to offsets within Mach memory objects. The primary role of the kernel in virtual memory management is to manage physical memory as a cache of the contents of memory objects. The kernel's representation for the backing storage of a memory object is a Mach port to which messages can be sent requesting or transmitting memory object data. Memory object backing store can thus be implemented by user-state programs such as the Unix Server. The Unix Server makes use of this facility to act as an ''external inode pager''.

Scheduling

Unix application processes are implemented by the Unix Server as Mach tasks with a single thread of control. These threads are scheduled by the Mach kernel using a multi-level feedback queue scheduler with 32 priority levels. Because all Unix application tasks are created by the Unix Server, the Unix Server has direct access to their task and thread ports. This allows it to directly manipulate the priority and schedulability of these tasks to approximate traditional Unix scheduling and priority management.

System call redirection

The Mach kernel allows a designated set of system calls or traps to be handled by code running in user mode within the calling task. The set of emulated system calls needs to be set up only once; it is inherited by child tasks on fork operations.

Device Support

The Mach kernel provides all low-level device support. Each device is represented as a port to which messages can be sent to transfer data or control the device. Data is transferred through read and write operations; the request and reply messages are exported separately, allowing both synchronous and asynchronous styles of I/O. The external memory object protocol allows a user to map the frame buffer for a graphics device directly into its address space.

User Multiprocessing

A user-level multithreading package, the C Thread library [3], eases the use of multiple threads within an address space. It exports mutual exclusion mutex locks and condition variables for synchronization via condition_wait and condition_signal operations.

Unix Server

The bulk of Unix services are provided by the Unix Server. It is implemented as a Mach task with multiple threads of control managed by the Mach C Threads package. Internal synchronization and process-switching within the Unix Server (e.g., sleep, wakeup, spl) are implemented by using the C Threads package's mutex, condition_wait and condition_signal functions. A typical system configuration will have dozens of C Threads allocated within the Unix Server. Most threads belong to a common pool which handle incoming requests from user processes. Several threads are dedicated to routines that, in a BSD kernel, would be driven by hardware interrupts (device IO completion, timeout, network code). All communication with hardware devices is done through Mach's IPC (Interprocess Communication) facility. Figure 3-1 shows the organization of the Unix Server and its relationship to the Mach kernel.

The primary tool for communication between the Unix Server and a Unix application program is Mach IPC. Most requests for service arrive in the form of Mach messages requesting that a Unix operation or service be performed. For each incoming message a C Thread is dispatched from the pool to handle that operation. That thread then determines which Unix process requires service, what operation is to be performed and finally parses the message to obtain the arguments for that operation. Many, but not all, messages to the Unix Server correspond directly to system calls normally present in 4.3BSD.

The main departure from this style of interaction between the Unix Server and a Unix application can be found in the handling of 4.3BSD file access. Access to Unix files can be provided either through a pure message passing interface or through the Mach memory object facility. The decision of which interface to use can be made either by the Emulation Library or the Unix Server. The primary reason to choose a pure message passing interface would be performance in a network environment where a message passing interface corresponds more precisely to the natural implementation technology of a network. In a tightly coupled multiprocessor or a uniprocessor a memory object interface is a more efficient way to transfer large amounts of data.

In the case of a memory object implementation of file access, the Unix Server acts as the memory object manager (or "inode pager") for 4.3BSD files. When a file is opened by a Unix application its data is mapped directly into the portion of the Unix application address space occupied by the Transparent

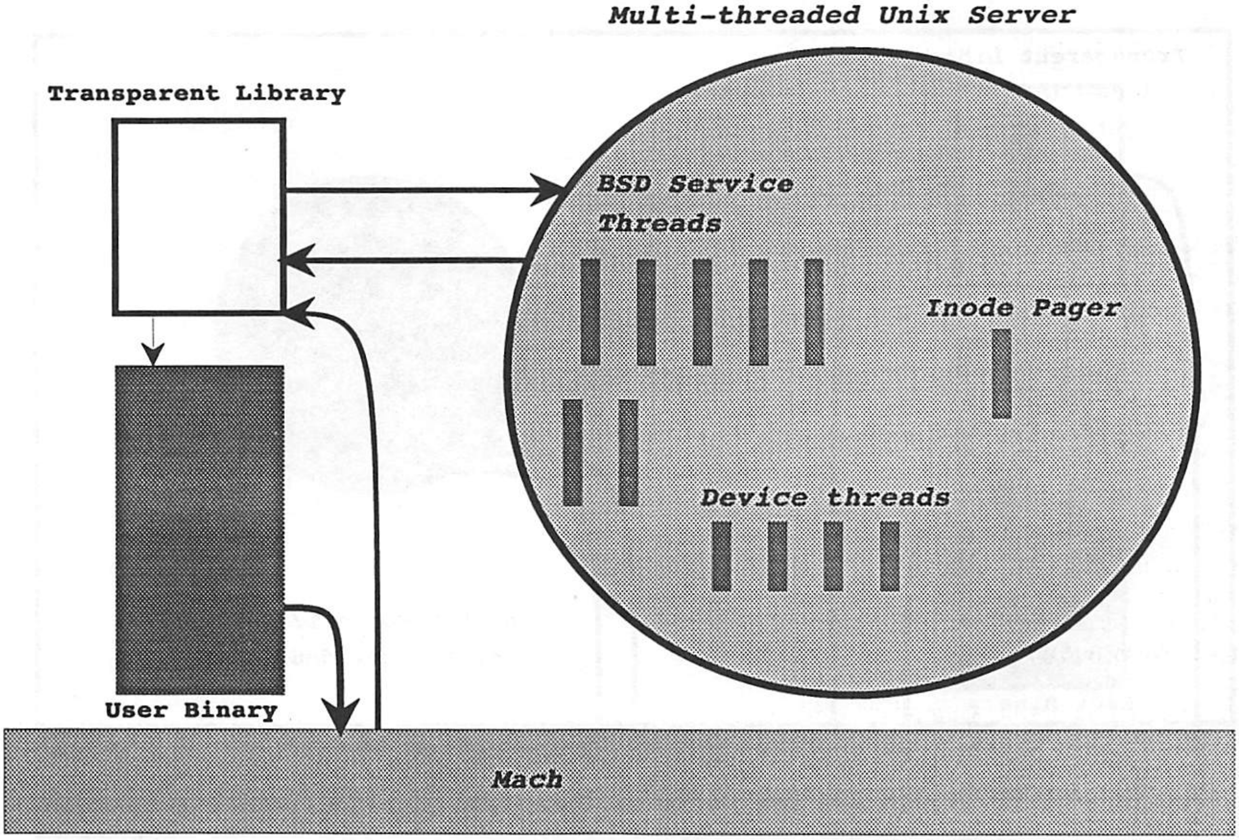


Figure 3-1: Organization of Unix services

Emulation Library. That library then directly provides read, write, lseek, etc. system call access to the file's data. In order to ensure Unix file sharing semantics, the Transparent Library must hand-shake with the Unix Server through messages whenever conflicts with other applications could arise. See figure 3-2.

Transparent Emulator Library

The Transparent System Call Emulation Library is loaded into the address space of the first user process (/etc/init). Mach allows memory to be inherited either read/write or copy-on-write from parent to child task. The Unix Server takes advantage of this function to allow the Transparent Library to be inherited by each child process from its parent on a fork operation. Execve and similar operations which reload an address space with a new application program are careful to preserve the Transparent Library portion of the address space. One advantage of this technique is that it allows multiple Transparent Libraries with perhaps different behavior (such as the support of somewhat different Unix variants) to co-exist with the same Unix Server.

The Transparent Emulation Library contains the equivalent of the Unix system call handler and the glue routines necessary to transform system calls into remote procedure calls to the Unix Server. It intercepts system calls using the Mach system call redirection facility and transforms them into remote procedure calls to the server process. Most of the machine dependencies in the BSD code are handled by the Library. These include manipulating the application's stack for signal handling and forking a new process.

In some cases Unix requests can be handled exclusively by the Transparent Library. For example, a Unix file which has been mapped into the Transparent Library memory region as part of an open call can be directly read by the Transparent Library without requiring the intervention of the Unix Server. Likewise some Unix signal management (e.g., sigvec), memory management (e.g., obreak) and state management (e.g., getpid) can be performed solely by the Emulation Library.

It is important to note that Transparent Emulation Library code is always executed with its own stack and does not use the stack of the application making the system call. This allows the Emulation Library to be compatible even with applications which may be doing their own stack management or providing their own lightweight process mechanisms.

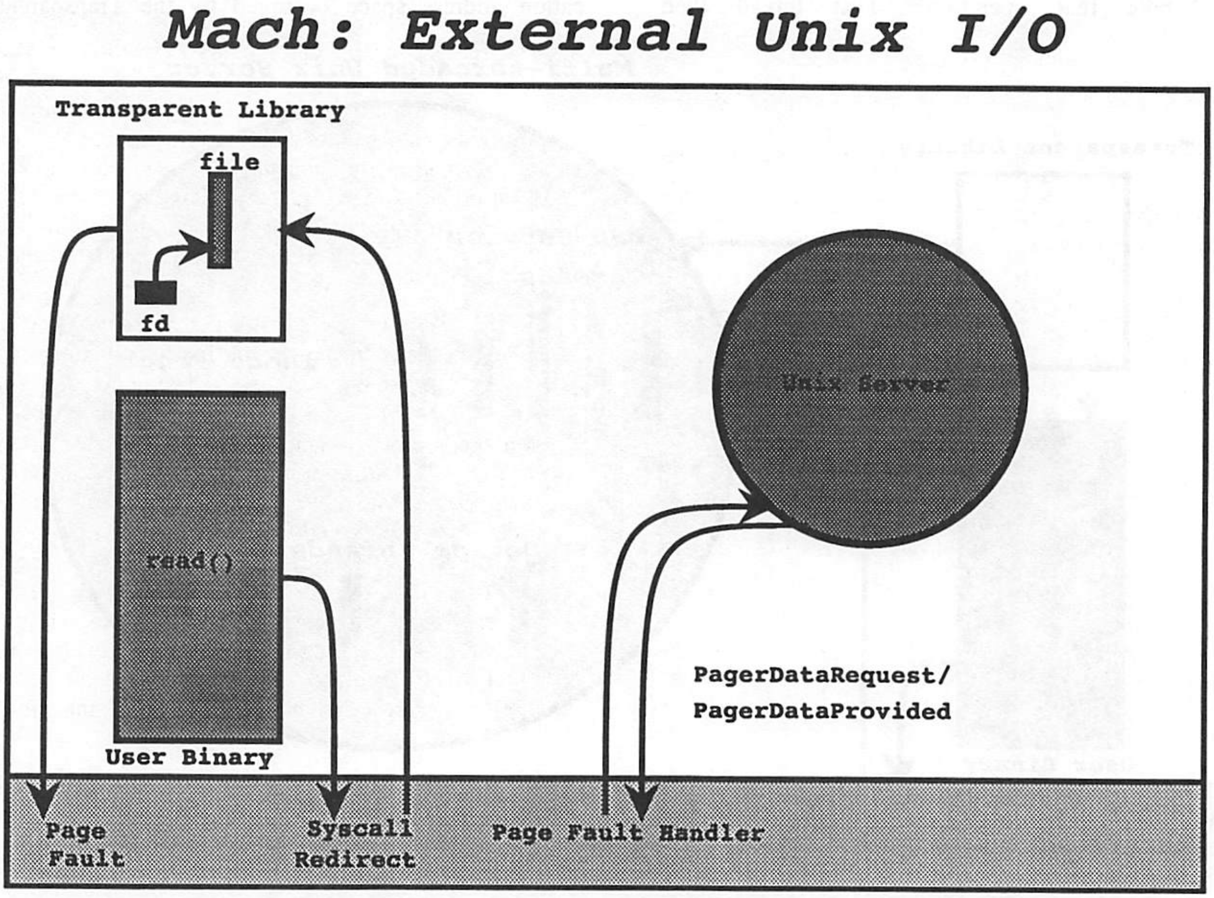


Figure 3-2: Implementation of Unix files as memory objects

It is also worth noting that the Transparent Emulation Library is vulnerable to tampering by an application program. It resides in a portion of the user application's address space and it is within the capabilities of a user program to either read/write or remove that region of memory. As a result, care has been taken to ensure that the correct functioning of the Unix Server cannot be affected by malicious or unintentional tampering with the Emulation Library. It is also important that information managed by the Transparent Library not be more security sensitive than information otherwise available to the user. In this regard the Transparent Library must operate under restrictions similar to that of the standard C runtime library.

Implementation History

We started the implementation of the Unix Server in January of 1989. The single-server Unix system ran multi-user by March. Much of the speed with which this was done can be attributed to the fact that we took as much advantage as possible of earlier Mach implementations. Although earlier versions of Mach – in particular Mach Release 2.5 – provided Unix compatibility in kernel state, they did so by redefining those operations in terms of Mach primitives. As a result, much of the code providing Unix services was already structured in a way which allowed its use outside of the Mach kernel.

The first versions of the Emulation Library and Unix Server converted every Unix system call to a message. The Mach memory object facility was only used to implement the text and data segments of application address spaces. The result was a system which performed better than we had anticipated but still needed to be tuned considerably. For example, a simple compilation test which would normally take 28.5 seconds to execute under Mach Release 2.5 on an 8MB Sun 3/60 ran, instead, in about 38 seconds.

Some of the problems of the Unix Server were in evidence in specific tests of system call and read/write I/O performance. The cost of a getpid call was about 1.4 milliseconds on a Sun 3/60 and an 8192 byte read operation was over 5 milliseconds. These compared with 118 microseconds for getpid and 2 milliseconds for a read on Mach 2.5.

We examined in detail the behavior of the system running a variety of benchmarks and determined four areas in which substantive improvements would have the greatest effect:

1. The Mach IPC facility,
2. The Mach C Thread library,
3. management of signals and other forms of shared Library/Server state,
4. management of Unix file data (read/write/lseek in particular).

Mach IPC performance enhancements

At the time we began our work, the performance of the Mach IPC facility was substantially better than other Unix communication facilities [1] but was not tuned to the kind of intense use required by the Unix Server. In March of 1989 the minimum round trip remote procedure call times (RPC) for Mach on the MicroVAX III was measured as 800 microseconds. On the Sun 3/60 minimum RPC times were 1100 microseconds. For a compilation test requiring approximately 25-30 seconds to compete and utilizing 2600 system calls, this amounted to approximately 10% of elapsed time.

There were two ways to attack this problem and we chose to do both. We sped up the IPC facility by approximately 30% through the use of handoff scheduling. By the summer of 1989 our MicroVAX III minimum RPC costs were 450 microseconds and our Sun 3/60 costs were 800 microseconds. In addition, we reduced the total number of messages to somewhat less half of all system calls (using techniques described below). The result of these enhancements was a considerable reduction in the total time devoted to minimum-path IPC costs.

C Thread library enhancements

We found two problems with the Unix Server's use of the C Thread library. First, the total number of C threads needed by the Unix Server was potentially much larger than the number of kernel threads it was reasonable to dedicate to it. Secondly, the cost of synchronization using the then existing version of the C thread mutex package was too high. Both problems were caused by a one-to-one binding of C threads to Mach kernel threads in that version of the C thread package.

We addressed these concerns by modifying the C thread package to detach the implementation of C threads from kernel threads. The package was changed to allow a specified pool of kernel threads to animate a much larger pool of C threads. Mutex, condition_wait and condition_signal operations were in turn modified to hand off execution from one C thread to another directly rather than attempt to use kernel scheduling facilities. The number of kernel threads used by the Unix Server could be set to allow any specified level of kernel managed multiprocessing. In addition, specific C threads were allowed to be bound to kernel threads where necessary, for example to act as device management threads.

Management of signals and shared state

Another problem area which became evident immediately upon instrumentation of the system was the frequent calls by Unix applications (especially the Unix shells) which are intended merely to set or read state information shared with the operating system. These calls include operations to modify signal handling state, verify the identity of the current process and modify the current address space by allocating new memory.

After some experimentation we determined that these operations could either be cached in a memory area accessible to both the Emulation Library and to the Unix Server or they could be performed directly by the Emulation Library using the underlying Mach kernel mechanisms. Thus, along with each Unix fork operation, the Unix Server allocates two memory objects which it maps into the newly created process. The first of these is a read-only object which contains information about the process which the Server is willing to communicate to it. The second is a read-write object accessible to both the Library and the Server. This object is used to allow the Emulation Library to update state information which is not immediately needed by the Server.

Although attractive on a tightly coupled multiprocessor or a uniprocessor, this use of shared state is not necessary appropriate to other architectures. We therefore made it optional with the version of the client Emulation Library. A full message passing interface to the server was preserved.

Management of Unix file data

One of the major costs in any Unix system is the cost of data movement to and from the disk subsystem. For example, we found that over 75% of all operations in a measured compilation suite were either open, close, read, lseek or write. Within this class, read, write and lseek dominated all file system operations.

The costs of data movement in turn dominate read and write system call times. On a Sun 3/60 the basic cost of an 8192 byte data copy is 1.2 milliseconds. Because Unix read and write system calls specify their target byte addresses it is not possible, in general, to use virtual memory page management to move data directly into a client address space. The simplest way to move 8K of data from the Unix Server to a client application is by Mach messages which actually contain 8192 bytes of copied data. Unfortunately this approach can result in as many as four data copies per call and minimum read or write times of greater than 5 milliseconds.

We experimented with a variety of techniques for reducing these data copy costs. Ultimately the best strategy was to effectively eliminate the cost of data copy altogether. This was accomplished by having the Unix Server map files directly into the address space managed by the Transparent Emulation Library. This allowed read, write and lseek operations to be performed completely by the Library rather than the Server. State information pertaining to the Library's access to a file is shared with the Unix Server through a shared memory object page (see the previous section).

The role of the Unix Server in this approach is that of a memory object manager for inodes and a synchronization manager for files which are shared by multiple simultaneously executing Unix processes. This eliminates the costs of message passing for many operations as well as the cost of extra data copies. Only one copy is ever made on a read or write call: the necessary copy from the application specified byte address to the actual page of the file in memory. Another advantage of this approach is greater potential parallelism. Because the operation is confined to the client application there is typically no lock contention or serialization.

The major drawback of this approach is increased costs for open and close operations on files. These operations can require memory mapping and deallocation costs which, in practice, are reasonably expensive. Our investigations have shown that open/close costs are important, but not nearly as important as the cost of read/write/lseek operations.

The results of our modifications to the file system have been gratifying. Instead of taking over 5 milliseconds for an 8192 byte lseek/read the costs for an lseek/read loop on the Sun 3/60 were reduced to less than 1.4 milliseconds. This compares favorably to the 2.12 milliseconds for the same test running under Mach Release 2.5.

Performance

We examined the performance of our multithreaded Unix Server and Emulation Library through the use of two system oriented benchmarks, one a simple file compilation benchmark and the other a more comprehensive file system test. We also timed specific system calls both individually and in frequently used combinations. In the following sections all tests were made running either Mach Release 2.5 or the Mach kernel with the Unix Server described in this paper. All tests were made on a Sun 3/60 with 8MB of memory and a Priam 300MB disk drive. The same disk with the same binaries and user environment was used in all tests.

A compilation test suite

We devised a relatively simple compilation test suite and used it to measure the performance of both systems. This compilation test consisted of a shell command file which ran nine compilations of small C source files. These files contained relatively few header file inclusions. Each compilation was separately timed using the ''time'' command.

The resulting test stressed process creation/termination, program load and startup, file open/close and read/write costs for small files. During the roughly 30 seconds required to complete the test it performed approximately 2600 Unix system calls, forked 57 processes, attempted to open 240 files and close 350 file descriptors, unlinked 100 files and called execve 160 times. Read, write and lseek operations accounted for a large fraction of all system calls. Roughly 750 lseek operations, 450 read and 230 write operations were performed. The table below shows the results of the benchmark. The runtime for SunOS 4.0 on the same machine is shown for purposes of comparison.

Compile Test Performance	
Operating System	Elapsed Time
SunOS 4.0	49 seconds
Mach Release 2.5	28.5 seconds
Mach w/Unix Server	28.4 seconds

File system benchmarks

In order to get a more complete evaluation of the system's file system performance we took advantage of a benchmark originally developed by M. Satyanarayanan for his performance evaluation of the Andrew File System [5]. Specifically we used a version of the Andrew Benchmark modified by John Ousterhout [7]. This benchmark stresses directory and file creation, file copy, file search (using ''find'') and compilation activity. A complete examination of this benchmark can be found in the cited papers. See Table 1 at the bottom of the page.

Probably the greatest differences revealed by this test were in the the costs of directory scanning and the ''stat'' operation. Neither have been examined or tuned in the current Unix server. The overall difference in performance was about three percent.

Basic File System Costs

In addition to our examination of various application benchmarks we looked at the cost of specific operations. In particular, we examined the cost of creating, writing and deleting files of various lengths, the throughput for both cached and uncached file read operations and for file write operations. The table below summarizes these results.

Overall, we found the behavior of the Mach w/Unix Server system to be very similar to that of our Mach 2.5 Release for these operations. In particular, the costs of cached read operations and of write operations were nearly identical.

File System Throughput		
Operation	Mach 2.5	w/Unix Server
create write (0 bytes) Delete	84.4ms	83.4ms
create write (10KB) delete	154.0ms	152.0ms
create write (100KB) delete	634.1ms	596.0ms
write (1MB)	0.26 MB/sec	0.27 MB/sec
read (cached)	3.98 MB/sec	3.93 MB/sec
read (uncached)	0.41 MB/sec	0.34 MB/sec

Measured separately, the costs for open, close, lseek and read operations expose the differences in implementation techniques and the tradeoffs between an in-kernel and out-of-kernel Unix environment. Repeated read and write operations of various lengths are considerably faster in our out-of-kernel system. The cost of an ''open'' call, however, has increased due to the required message handshakes and to the cost of memory mapping the referenced inode. See Table 2.

Process Management Costs

Our raw measurements of process creation and termination demonstrate a problem we have not yet resolved with the use of the Transparent Emulation Library. As can be seen from the table below, fork and exec costs range from 2 to 3 times greater for the Mach w/Unix Server than for Mach Release 2.5. The cost of the fork operation itself is the primary culprit. See Table 3.

The explanation for this increase in fork cost can be found in a large number of additional page faults which result from the use the Transparent Library. At fork, a copy-on-write copy of the parent is created. In a normal in-kernel Unix implementation, the parent modifies a single stack page (resulting in a copy-on-write fault) and then does a Unix ''wait'' for the child to exit. In our out-of-kernel Unix environment both the parent and child processes modify the top-of-stack page of their Transparent Library as well as their own process top-of-stack page. In addition references are made to

Modified Andrew Benchmark						
Operating System	Directory Creation	File Copy	Recursive file stats	Find	Compile	Elapsed
Mach Release 2.5	4 sec	20 sec	13 sec	26 sec	336 sec	399 sec
Mach w/Unix Server	1 sec	20 sec	24 sec	34 sec	332 sec	411 sec

Table 1

various code and data pages in the Transparent Library. As a result, twenty faults are taken in the out-of-kernel version. Of these faults, five are copy-on-write faults and three are fill zero faults. This contrasts with only five faults altogether for the in-kernel Unix of which two are copy-on-write faults. The high cost of program execution in the early SunOS 4.0 system (included for comparison) was similarly the result of its use of shared libraries.

Status

All 4.3BSD binaries, Ultrix binaries or Sun 3.x binaries which ran on earlier versions of Mach continue to run. The switch from older releases of Mach (or Ultrix on the DECStation) can be accomplished simply by installing a new kernel, the Unix Server and Emulator Library on an existing disk. The system includes support for X11R4, full Berkeley networking, the Andrew Nationwide File System and CMU's internal remote file system.

At the time of this writing (April, 1990) the pure Mach kernel with Unix as an application program was running on multiprocessor and uniprocessor Vax systems, the DECStation 3100, the Sun 3 and i386 PC-clones. We are working with the Open Software Foundation on a version for the Encore Multimax. We expect to support the Macintosh and other machines in the future.

The multithreaded Unix Server described in this paper is but one of two ongoing implementations of Unix functionality at CMU. A second group [9] has been engaged in an implementation of Unix based on multiple Mach server tasks each of which performs specific functions such as file access, authentication, etc. The advantage of this approach over a single server Unix implementation is flexibility (since each server can provide services for various operating system environments) and security (since each can be individually secured and verified).

Work using Mach to implement non-Unix environments is also in progress. A virtual Macintosh OS environment is already running on top of Mach Release 2.5 at CMU and is scheduled to be ported to the pure kernel. We are also providing support for DOS applications on the i386.

Conclusions

We believe we have demonstrated the practicality of implementing Unix as an application program. Our performance measures demonstrate differences in the underlying cost profiles for our in-kernel and out-of-kernel Unix systems but do not indicate serious problems. We continue to make improvements but we have already achieved near parity with Mach Release 2.5 and can outperform some commercial Unix implementations.

File System Operation Costs		
Operation	Mach Release 2.5	Mach w/Unix Server
lseek	160us	170us
lseek + read (1 byte)	0.44ms	0.35ms
lseek + read (8K bytes)	2.12ms	1.38ms
open "foo" + read (8K) + close	3ms	5.5ms
open "/usr/rfr/t/foo" + read (8K) + close	5.2ms	6.68ms
failed open of "xxx"	1.66ms	1.42ms

Table 2

Process Management Costs			
Operation	Mach Release 2.5	Mach w/Unix Server	SunOS 4.0
getpid	118us	102us	75us
fork + exit	16ms	48ms	27.6ms
fork + wait + exec + exit	32ms	74ms	106ms

Table 3

References

[1] Accetta, M.J., Baron, R.V., Bolosky, W., Golub, D.B., Rashid, R.F., Tevanian, A., and Young, M.W. *Mach: A New Kernel Foundation for UNIX Development.* In Proceedings of Summer Usenix. July, 1986.

[2] Armand F., Gien M., Guillemont, M. and Leonard, P. *Towards a Distributed UNIX System - The CHORUS Approach.* In Proceedings of the European UNIX Systems User Group Conference. September, 1986.

[3] [3] Cooper, E. and Draves, R. *C Threads.* Technical Report CMU-CS-88-154, Computer Science Department, Carnegie Mellon University, June, 1988.

[4] [4] Parmelee, R. P, T. I. Peterson, C. C. Tillman and D. J. Hatfield. *Virtual Storage and Virtual Machine Concepts. IBM Systems Journal 11(2):99-130, 1972.*

[5] Howard, J.H., Kazar, M.L., Menees, S.G., Nichols, D.A., Satyanarayanan, M., Sidebotham, R.N., West, M.J. Scale and Performance in a Distributed File System. ACM Transactions on Computer Systems 6(1), February, 1988.

[6] Lycklama, H. and Bayer, D. L. *The MERT Operating System.* Bell System Technical Journal , July, 1978.

[7] Ousterhout, J. *Why Aren't Operating Systems Getting Faster as Fast as Hardware?* In Proceedings of Summer Usenix. June, 1990.

[8] Thacker, C. P. and Stewart, L. C. and Satterthwaite, Jr., E. H. *Firefly: A Multiprocessor Workstation.* IEEE Transactions on Computers 37(8):909-920, August, 1988.

[9] Rashid, R., Baron, R., Forin A., Golub, D., Jones, M., Julin, D., Orr, D., Sanzi, R. *Mach: A Foundation for Open Systems; a Position Paper.* In Proceedings of the 2nd Workshop on Workstation Operating Systems. IEEE, September, 1989.

David Golub is a Senior Research Programmer and has worked on the Mach project since December 1985. He received the B. A. in mathematics from Brown University in 1975. His research interests are in distributed computing.

Randall Dean is a Research Systems Programmer working with the Mach Project since May, 1989. His primary interests are rock climbing and volleyball.

Dr. Forin received the B.S. degree in Electrical Engineering in 1982 and the Ph.D. degree in Computer Science in 1987, both from the University of Padova, Padova, Italy. He is currently a Research Computer Scientist at the Carnegie Mellon University. Dr. Forin is member of the Association for Computing Machinery, SIGPLAN and SIGOPS.

Professor Rashid is Director of the Mach Project and has been on the faculty of Carnegie-Mellon University since September, 1979. He received his M.S. (1977) and Ph.D. (1980) degrees in Computer Science from the University of Rochester. He had previously graduated with Honors in Mathematics from Stanford University (1974).

References

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

USENIX Summer Conference
June 11-15, 1990
Anaheim, California

Mark Heuser – Harris Computer Systems

An Implementation of Real-Time Thread Synchronization

ABSTRACT

High-priority threads need fast and predictable service in real-time applications. When threads interact by serializing their access to shared data, it is possible for one or more low-priority threads to indefinitely postpone the execution of a high-priority thread. This situation, known as unbounded priority inversion, poses a serious threat to the application's ability to meet its deadlines. The delay experienced by high-priority threads in the presence of contention must be bounded, and synchronization overhead in the absence of contention should be low. This paper describes a collection of low-level synchronization tools – CPU rescheduling control, busy-wait mutual exclusion, and client-server coordination – designed to meet these needs in a UNIX operating system. Together they can be used to implement basic priority inheritance, the simplest of the techniques for controlling priority inversion. The effects of priority inheritance and unbounded priority inversion are demonstrated with some measurements.

Introduction

Many real-time applications are structured as a collection of cooperating, concurrently executing tasks, even when the underlying hardware does not support true parallelism. Task execution may be triggered by external or internal events at regular or irregular intervals. Tasks may have different arrival rates, resource requirements, and timing constraints. Real-time scheduling theory is concerned with determining whether a given set of tasks can be scheduled so as to meet their deadlines. In 1973, Liu and Layland derived the *rate monotonic scheduling algorithm* and a sufficient condition under which the algorithm will successfully schedule a set of independent, periodic tasks on a uniprocessor. The algorithm is a preemptive, static-priority scheduling algorithm that assigns higher priorities to tasks with smaller periods. They showed that the algorithm is optimal in the sense that if a set of tasks can be scheduled by any other static-priority scheduling algorithm, they can also be scheduled by the rate monotonic algorithm. Since then, many researchers have built on Liu and Layland's original work.

In practice, tasks are often not independent; they need to serialize their access to shared data. It is possible for one or more low-priority tasks to indefinitely postpone the execution of a high-priority task – a situation known as *unbounded priority inversion.* In 1987, Sha et al. proposed techniques for controlling priority inversion. Given a bound on priority inversion, they derived a sufficient condition under which interacting tasks can be scheduled by the rate monotonic scheduling algorithm.

This paper describes a collection of low-level synchronization tools for a UNIX operating system. Together they can be used to implement *basic priority inheritance*, the simplest of the techniques for controlling priority inversion. Other techniques for controlling priority inversion, such as the *priority ceiling protocol* [Sha et al., 1987], provide better worst-case bounds but are more difficult to implement. In the development of these tools, emphasis was placed on low synchronization overhead in the absence of contention and controlled priority inversion in the presence of contention.

In other related work, Edler et al. [1985] used similar synchronization tools on the NYU Ultracomputer but did not address priority inversion. Luttmer et al. [1989] implemented priority inheritance in a real-time executive as opposed to a UNIX operating system.

Section 2 surveys some of the issues involved in synchronizing a set of real-time tasks. It discusses the use of shared memory for synchronization, busy-wait vs. sleepy-wait mutual exclusion, the definition of priority inheritance, concerns about portability, and the wake-up waiting race condition. Section 3 describes the synchronization tools themselves and provides some examples of their use. The tools are grouped into three categories: CPU rescheduling control, busy-wait mutual exclusion, and client-server coordination. Section 4 provides a few implementation details. Finally, Section 5 evaluates the tools and demonstrates the effects of priority inheritance and unbounded priority inversion with some measurements. The reader is assumed to have a general familiarity with task synchronization, to which Birrell [1989] provides an excellent introduction.

Background

For the purposes of this paper, the term *thread* refers to a thread of execution that shares some or all of its address space with other threads of execution.[1] The essential characteristic of a thread is its shared address space, not its ''weight.'' For example, UNIX processes attached to the same shared memory segment are considered threads, as are lightweight processes sharing an entire address space. In either case, two or more threads of execution, possibly running on different CPUs at the same time, need to coordinate their access to the shared address space in an efficient and deterministic manner.

Shared Memory Considerations

Shared-memory critical sections are often very short. To keep the cost of synchronization comparatively small, synchronizing operations performed on entry to and exit from a critical section need to be fast. In particular, they should avoid entering the kernel whenever possible. No kernel intervention should be required to enter an unlocked critical section or to leave a critical section when there are no waiting threads. Synchronizing variables can reside in the same address space as the shared data, where they are directly accessible to the application. Contrast this situation to one wherein the interacting tasks are not assumed to share address space. In that case, the synchronizing variables must reside in the kernel and can be manipulated only with system calls (e.g.; AT&T System V IPC semaphores, P1003.4 binary 5 semaphores).

Short critical sections suggest a small delay when a thread arrives at a locked mutex. When the expected delay is smaller than the cost of a context switch, busy-waiting (spinning) should be a viable alternative to sleepy-waiting (blocking). Unfortunately, a critical section may be occupied far longer than expected due to such unpredictable events as page faults, signals, and CPU rescheduling. At best, these events may cause other CPUs to delay longer than anticipated; at worst, they may cause deadlock. Some of these events can be controlled with system calls, but system call overhead is usually prohibitive.

Real-Time Considerations

Priority inversion occurs when one or more low-priority threads prevent the progress of a high-priority thread. It happens in the context of mutual exclusion and is particularly aggravated by the static-priority scheduling disciplines found in many real-time systems. Suppose, for example, that a low-priority thread L is preempted inside a critical region. A high-priority thread H then attempts to enter the critical region and blocks. H cannot continue until L leaves the critical region, and threads in the priority gap (see Figure 1) between H and L prevent L from running. H effectively assumes L's priority and blocks for unpredictable and possibly lengthy periods of time.

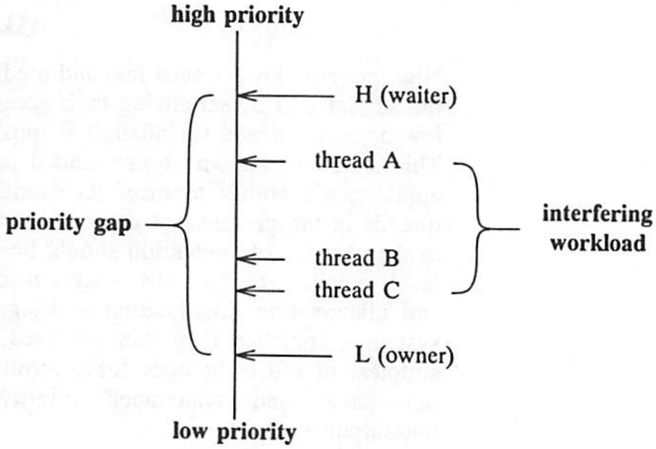


Figure 1. Priority Inversion

Raising L's priority when it enters the critical section would help to prevent unbounded priority inversion. If the priority alteration is inexpensive, this method may work well in cases of busy-wait mutual exclusion where critical sections are short. However, if the critical section is long enough to warrant use of sleepy-wait mutual exclusion, this approach may be too conservative. It will defer the preemption of L in cases where no inversion would have occurred.

Raising L's priority to that of H when H blocks on entry to the critical section would help to correct the inversion when it occurred. This approach has been called *basic priority inheritance* by Sha et al. [1987]. It closes the priority gap long enough to allow L to leave the critical section. The following more formal definition of priority inheritance is adapted from Luttmer et al. [1989]:

"*Running Thread Invariant (RTI)*"

The running threads are the runnable threads with the highest effective priorities.

"*Effective Priority*"

A thread's effective priority is the maximum of its assigned and inherited priorities.

"*Assigned Priority*"

A thread's assigned priority is the priority established by the thread's scheduling policy (static-priority, nearest-deadline, and so on).

"*Inherited Priority*"

A thread's inherited priority is the maximum effective priority of all those threads blocked on entry to critical sections it occupies. If a

[1] This use of the term is more general than is commonly accepted.

thread is not blocking the execution of other threads, its inherited priority is the lowest possible priority value.

Note that the definition of effective priority is recursive. A thread's effective priority is defined in terms of its assigned priority and the effective priorities of other threads. In other words, priority inheritance is transitive. If a high-priority thread H were blocked on a critical section occupied by medium-priority thread M, and if M were blocked on a different critical section occupied by low-priority thread L, both M and L should inherit H's priority.

Using a model similar to that in Luttmer et al. [1989], the relationships between threads and mutexes can be represented by a directed bipartite graph G = (T, M, W, O). T and M are sets of vertices that represent threads and mutexes, respectively. W is a set of "wait-for" arcs from T to M that indicate which threads are blocked on which mutexes. O is a set of "owned-by" arcs from M to T that indicate which mutexes are locked by which threads. Figure 2 illustrates a graph containing six threads and three mutexes. Threads t3 and t4 are blocked on mutex m3, which is locked by thread t6. Thread t3 had previously locked mutex m1, which is now blocking threads t1 and t2. Mutex m2 is locked by thread t5.

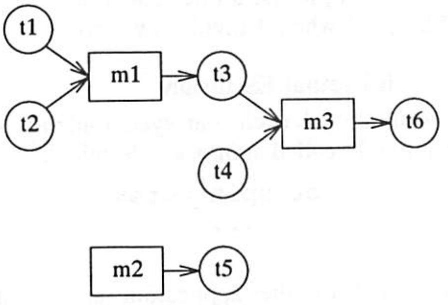


Figure 2. A graph G

The graph changes over time as the relationships between threads and mutexes change. Changes to a thread's inherited and effective priorities can be related to changes in the graph. When a change invalidates the RTI, corrective action must be taken.

Creation of an Owned-By Arc
An owned-by arc is created when a running thread locks an unlocked mutex. If there are no waiting threads, creation of an owned-by arc has no effect on priorities. If there are waiting threads, creation of an owned-by arc may increase the running thread's effective priority.[2] The RTI is preserved in either case.

Creation of a Wait-For Arc
A wait-for arc is created when a running thread attempts to lock a locked mutex. The new wait-for arc may increase the effective priorities of downstream threads and invalidate the RTI. Because the running thread is blocking, rescheduling is required on the current CPU.

Destruction of an Owned-By Arc
An owned-by arc is destroyed when a running thread unlocks a mutex. If there are no waiters, destruction of an owned-by arc has no effect on priorities. If there are waiters, destruction of an owned-by arc may decrease the running thread's effective priority and invalidate the RTI. This event coincides with the destruction of at least one wait-for arc because at least one of the waiting threads will be awakened when the mutex is unlocked. LP *Destruction of a Wait-For Arc*
A wait-for arc is destroyed when a running thread unlocks a mutex. It makes a new thread runnable and may invalidate the RTI. It may also occur if a thread's wait is terminated prematurely, perhaps in response to a signal or a timeout. In this case, it may decrease the effective priorities of downstream threads and invalidate the RTI.

The graph analysis suggests that mutexes must be "owned" rather than simply "locked." Synchronizing routines need to record more than the binary locked/unlocked status of a critical section; they need to record the actual identity of the thread in a critical section. The graph also suggests that most of the changes to inherited and effective priorities and most invalidations of the RTI can be associated with the creation or destruction of wait-for arcs. Locking an unlocked mutex and unlocking a mutex that has no waiters – the creation and destruction of owned-by arcs – remain simple operations even in the context of priority inheritance and should not require kernel intervention.

Other Considerations

A thread may lock a mutex to inspect some shared data and then decide to block until the shared data change state. Before blocking, the thread must release the mutex to allow access to the shared data by other threads. As soon as the mutex is released, the state of the shared data may change. Special care must be taken to ensure that the thread will not miss a wake-up that occurs after it releases the mutex and before it actually blocks. The system calls used to block and later to awaken the thread are influenced by the approach taken to eliminate this *wake-up waiting race condition.* The MACH system call interface exports three user-level thread states and a collection of state transition primitives [Tevanian et al., 1987]. Although the interface is very clean, two system calls (state transitions) are required to block the thread and the mutex must remain locked during one of the system calls.

[2] Waiting threads might exist if only one is awakened each time a mutex is unlocked.

Another approach is to queue at most one wake-up sent to a running thread and to apply it to the thread's next request to block. This approach blocks the thread with just one system call, but it requires that the thread be identified by name when it is awakened. If the thread were to block on a kernel-provided queue, and if members of the queue were not individually named when they were awakened, another approach would be required. A system call might be defined, for example, to awaken the highest priority thread in the queue with no prior knowledge of the thread's identity. For this last case, one could add the mutex to the blocking system call's interface. The system call would be responsible for releasing the mutex and blocking its caller atomically. Examples of these situations appear later in the paper.

Yet another consideration is portability. Architectural support for synchronization can take many different forms: fetch-and-increment, compare-and-swap, and swap are but a few of the synchronizing instructions found in modern CPUs. The tools presented in this paper make no attempt to exploit the raw computational power of some of the more complex instructions. To ensure portability, they rely on a simple test-and-set capability only. A test-and-set primitive atomically sets a location in memory to some nonzero value and returns the previous state of that location.

Interfaces

This section describes the synchronization tools themselves and provides some examples of their use. There are tools to prevent CPU rescheduling, provide busy-wait mutual exclusion, and coordinate threads interacting as clients and servers.

Rescheduling Control

Rescheduling variables provide low-overhead control of CPU rescheduling and signal delivery. A rescheduling variable is a data structure

```
struct resched_var {
    pid_t rv_pid;
    tid_t rv_tid;
    ...
};
```

that controls a single thread's vulnerability to rescheduling. It is allocated on a per-thread basis by the application, not by the kernel. A system call informs the kernel of the location of the variable, and the kernel examines the variable before making rescheduling decisions. Direct manipulation of the variable from user mode disables and re-enables rescheduling, resulting in very low overhead. These operatons are summarized in Table 1. For convenience, a rescheduling variable contains the owning thread's thread ID and process ID. The remaining fields are not meant to be referenced directly by the application.

Rescheduling locks may be nested. As long as a thread does not enter the kernel, quantum expirations, preemptions, and some signal deliveries are deferred until all of the rescheduling locks are released. No guarantees are made if a thread enters the kernel – through a page fault or system call, for example. In those cases, a thread may receive signals or lose control of the CPU regardless of the number of rescheduling locks it holds. Furthermore, signals representing error conditions are not affected by rescheduling locks: a thread will always receive a SIGFPE signal when it divides by zero.

Busy-Wait Mutual Exclusion

Busy-wait mutual exclusion synchronizing variables are commonly called *spinlocks*. Spinlocks

```
struct spin_mutex {
    ...
};
```

are allocated by the application and reside in a shared portion of a thread's address space. The interfaces in Table 2 operate on spinlocks.

Interface	Type	Description
resched_cntl(cmd, arg)	system call	Informs the kernel of the location of the rescheduling variable.
resched_nlocks(r)	macro	Returns the number of rescheduling locks in effect.
resched_lock(r)	macro	Increments the number of rescheduling locks.
resched_unlock(r)	macro	Decrements the number of rescheduling locks.

Table 1: Rescheduling Control Interfaces

Note that there is no interface to unconditionally lock a spinlock. A surprising number of issues surrounds an unconditional lock. An application may want to terminate a lengthy busy-wait to enter a debug mode. An application may want to measure arrival rates, contention rates, wait time, hold time, and so on. Anderson [1990] states that the performance of some applications is affected by the method used to poll a locked spinlock. The resolution of these issues is left to the application or library.

To illustrate how the interfaces presented in Tables 1 and 2 might be used, the macros below acquire and release spinlocks. They contain no system calls and no procedure calls. CPU rescheduling is disabled to prevent unpredictable preemptions in the critical section. The appropriate portions of the address space are assumed to be locked into memory to prevent unpredictable page faults in the critical section. *_m* points to a spinlock in a shared portion of the address space; *_r* points to the running thread's rescheduling variable.

```
#define spin_acquire(_m, _r) \
{ \
  resched_lock (_r); \
  while (! spin_trylock (_m)) { \
    resched_unlock (_r); \
    while (spin_islock (_m)); \
      resched_lock (_r); \
  } \
}

#define spin_release(_m, _r) \
{ \
  spin_unlock (_m); \
  resched_unlock (_r); \
}
```

When a thread finds a spinlock locked, it waits for the lock to clear with *spin_islock*. In a shared-bus multiprocessor with private, coherent, data caches, this sequence is much more efficient than polling directly with *spin_trylock*. *Spin_trylock* contains an atomic test-and-set instruction that bypasses the data cache to manipulate memory across the shared backplane. *Spin_islock* reads from the per-processor data cache and generates very little bus traffic. Note also that CPU rescheduling is re-enabled around *spin_islock* so as not to defer rescheduling any longer than necessary.

Client-Server Coordination

When one thread requests service from another, the former thread is called a *client*, and the latter thread is called a *server*. A client will usually wait for a response from its server before continuing execution. When a server completes a request, it awakens the corresponding client and begins work on the next request. If no other requests are pending, a server blocks to await the arrival of the next request. These interactions are typical of an Ada rendezvous or a remote procedure call.

A server should inherit its client's priority. If the implicit dependency of a client on its server were made explicit, the operating system would have the information necessary to implement priority inheritance. Suppose every thread possessed a queue of blocked clients. To await a response from its server, a client would block on the server's queue and, in so doing, identify the source of the response – the server – to the operating system. The interfaces in Table 3 formalize this interaction.

Client_block blocks the calling thread on the *server*'s queue and passes the caller's effective priority to the *server*. At any point in time, a thread has an effective priority at least as high as any of its blocked clients.

The *chan* parameter of *client_block* is simply an integer value used to group clients. If several clients were to wait for the occurrence of the same event, they would group themselves under a single value of *chan*. The server would then use *client_wakechan* to wake the entire group. *Client_wake1* wakes the single specified *client*.

Client_block handles the wake-up waiting race by releasing spinlock *m* and blocking the calling thread atomically. *Client_wake1* and *client_wakechan* only

Interface	Type	Description
spin_init(m, nm)	procedure	Initializes a collection of spinlocks.
spin_trylock(m)	compiler intrinsic	Attempts to lock a spinlock. Returns true if successful; otherwise false.
spin_islock(m)	macro	Returns true if a spinlock is locked; otherwise false.
spin_unlock(m)	macro	Unlocks a spinlock.

Table 2: Busy-Wait Mutual Exclusion Interfaces

wake threads that are actually blocked in *client_block*.

The interfaces in Table 3 manipulate threads acting as clients. When a server blocks to await the arrival of the next request for service, it has no dependencies on other threads. The client interfaces would, therefore, be inappropriate for the server to use. The interfaces in Table 4 manipulate threads acting as servers.

Server_block blocks the calling thread if no wake-up has occurred since the last return from *server_block*. It does not establish any scheduling dependencies on other threads. *Server_wake1* and *server_wakevec* wake threads blocked in *server_block*, queueing at most one wake-up sent to a running thread. A queued wake-up is applied to the thread's next call to *server_block*. Note that an option to *client_block* may also awaken a server.

All of the client-server routines accept an optional rescheduling variable, *r*, as an argument. If specified, the number of rescheduling locks in the variable is decremented. Releasing a rescheduling lock, blocking the running thread, and waking a blocked thread are all events that may reschedule a CPU. Coupling the release of a rescheduling lock with the blocking or waking of a thread eliminates extraneous rescheduling that could occur if they were performed as two separate actions.

To illustrate how the client interfaces might be used, the code fragments below implement priority inheritance in sleepy-wait mutual exclusion. Let

```
    struct sleep_mutex {
        struct spin_mutex mx;
        tid_t owner;
        int waiters;
    };
```

represent a sleepy-wait mutex. The *owner* field identifies the thread that owns the mutex, the *waiters* field indicates whether threads are blocked on the mutex, and the *mx* field serializes access to the mutex. Let *rv* represent the running thread's rescheduling variable.

A mutex may be locked as follows.

```
void
sleep_lock (s)
    struct sleep_mutex *s;
{
    spin_acquire (&s->mx, &rv);
    while (s->owner != 0) {
        s->waiters = 1;
        client_block (s->owner, s, 0,
                &s->mx, &rv, 0);
        spin_acquire (&s->mx, &rv);
    }
    s->owner = rv.rv_tid;
    spin_release (&s->mx, &rv);
}
```

Although perhaps an unusual interpretation of the client-server relationship, the owner of the mutex is considered to be a server, and the waiting threads are considered to be its clients. *Client_block* guarantees that the owner's priority is at least as high as any of the waiting threads. Note that the address of the mutex is used as the *chan* argument to

Interface	Type	Description
client_block(server, chan, options, m, r, timeout)	system call	Blocks the calling client and optionally wakes the server.
client_wake1(server, client, r)	system call	Wakes the specified client.
client_wakechan(server, chan, r)	system call	Wakes the specified group of clients.

Table 3: client Interfaces

Interface	Type	Description
server_block(options, r, timeout)	system call	Blocks the calling server.
server_wake1(server, r)	system call	Wakes the specified server.
server_wakevec(servers, nservers, r)	system call	Wakes the specified vector of servers.

Table 4: Server Interfaces

client_block to distinguish one mutex's waiters from another's.

A mutex may be unlocked as follows.

```
void
sleep_unlock (s)
    struct sleep_mutex *s;
{
    int were_waiters;

    spin_acquire (&s->mx, &rv);
    if (s->owner != rv.rv_tid) {
        /* Indicate error condition. */
    }
    s->owner = 0;
    were_waiters = s->waiters;
    s->waiters = 0;
    spin_unlock (&s->mx);

    if (were_waiters)
        client_wakechan(rv.rv_tid,s,&rv);
    else
        resched_unlock (&rv);
}
```

When the owner releases the mutex, all of the waiting threads are awakened. The newly awakened waiters recontend for the mutex when they execute. One will become the new owner of the mutex, and the others will block to establish new client-server relationships.

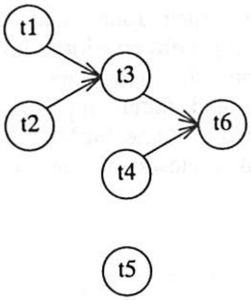


Figure 3: A graph G′

Relating this example to the graph discussed earlier, the set M of mutexes and the set O of owned-by arcs lie entirely within the domain of the application. Owned-by arcs are created and destroyed by changes to the *owner* field of a mutex; the operating system has no knowledge of their existence. From the application's point of view, one thread depends on another indirectly through a mutex. From the operating system's point of view, one thread depends on another directly. The kernel-level dependencies may be modeled with a directed graph G′ = (T′, W′), where T′ is the set T of threads in G, and W′ is a set of wait-for arcs from T′ to T′. Figure 3 illustrates the graph G′ that corresponds to the graph G shown in Figure 2.

In the example, note how the identity of the server changes each time a mutex is unlocked, forcing all waiters to update their wait-for arcs in W′. Waking all of the waiters may be inefficient on a multiprocessor. If long wait queues cannot be avoided, an Ada rendezvous style of mutual exclusion may be more efficient and can be implemented with the client interfaces in Table 3. The identity of the server remains constant in an Ada rendezvous, so only one client need be awakened upon exit from a critical section.

Implementation

The interfaces presented in Section 3 have been implemented in the CX/RT operating system, release 5.0. CX/RT is a fully preemptable, symmetric multiprocessor, UNIX compatible operating system with several real-time extensions. It currently runs on the Night Hawk platform: a tightly-coupled, shared-bus multiprocessor containing up to 8 MC68030 CPUs, per-processor external caches and local memories, and a large global memory.

To implement priority inheritance, CX/RT maintains explicit representations of the graph G′ and each thread's effective, inherited, and assigned priorities. Each thread possesses an eventcount [Reed et al. 1979] that contains a list of the thread's blocked clients; each blocked client identifies its server. *Client_block* adds the calling thread to the server's list of blocked clients and traverses the wait-for arcs to propagate the client's effective priority to downstream threads. The graph traversal stops as soon as it finds a thread with an inherited or effective priority greater than the client's effective priority or after it finds a thread not blocked in *client_block*. *Client_wake1* and *client_wakechan* remove the appropriate threads from the server's list of blocked clients and adjust the server's inherited priority to be the maximum effective priority of the remaining clients.

Evaluation

One of the goals of this work was low synchronization overhead in the absence of contention. When there is no contention, *spin_acquire* and *spin_release* together cost 8 microseconds on a Night Hawk 3800. *Sleep_lock* and *sleep_unlock* together cost 30 microseconds.[3]

Another goal was controlled priority inversion in the presence of contention. The following experiment was used to measure the effects of priority inheritance. A high-priority thread H and a low-priority thread L shared a mutex. When they were activated, both threads would spin for a small time, lock the

[3] These times are for a 20MHz MC68030 with a cold cache.

mutex, spin again, unlock the mutex, spin again, and terminate. A medium-priority thread M would merely spin for a while and terminate. L was allowed to execute first and lock the mutex. H and M became runnable while L was in the critical section. H preempted L, attempted to lock the mutex, and blocked. H's delay was measured as the execution time of M varied. The experiment was performed both with and without priority inheritance.

For the results shown in Figure 4, H's and L's spins each lasted approximately 375 microseconds. M's spin varied from 0 to 2500 microseconds. Instrumented versions of *sleep_lock* and *sleep_unlock* were used to obtain the priority inheritance measurements. They were reimplemented with the server interfaces in Table 4 to obtain the unbounded priority inversion measurements.

The measurements show that without priority inheritance both M and L contributed to H's delay. With priority inheritance, H's delay hovered at 700 microseconds, unaffected by M.

The most significant drawback to this implementation of priority inheritance is the requirement that all waiting threads be awakened when a mutex is unlocked. As indicated previously, if long wait queues cannot be avoided, an Ada rendezvous style of mutual exclusion may be more efficient. Further investigation is needed to handle long wait queues more efficiently.

Basic priority inheritance bounds the delay experienced by a high-priority thread at a mutex; however, a thread that locks several mutexes may still delay at each individual mutex. Other techniques for controlling priority inversion, such as the priority ceiling protocol, guarantee that a high-priority thread will block on at most one mutex. Their implementation depends on a close relationship between the mutex manager and the thread scheduler. In contrast, the implementation of priority inheritance described in this paper separates the mutex manager and the thread scheduler as much as possible to reduce synchronization overhead. The cost of kernel entry and exit is a major obstacle to a low-overhead implementation. Further investigation is needed to realize smaller bounds on priority inversion.

Conclusions

High-priority threads need fast and predictable service in real-time applications. When threads interact by serializing their access to shared data, unbounded priority inversion poses a serious threat to the application's ability to meet its deadlines. This paper demonstrates that priority inversion can be controlled with small sacrifices in overhead. This is accomplished through the combined use of several low-level synchronization tools. Page locking and preemption locking are used to bound priority inversions in busy-wait mutual exclusion. Busy-wait mutual exclusion and system calls that contain thread dependency information are used to bound priority inversions in sleepy-wait mutual exclusion. Layering mutual exclusion at the application level and thread scheduling at the kernel level meets both the goals of low synchronization overhead in the absence of contention and controlled priority inversion in the presence of contention.

Acknowledgements

Many thanks to Bruce Jones and Joe Gwinn for several enlightening conversations during the course of this development. They gave both valuable technical insight and much appreciated encouragement. Thanks also to Anne McMichael for her meticulous and kind reviews of earlier drafts of the paper.

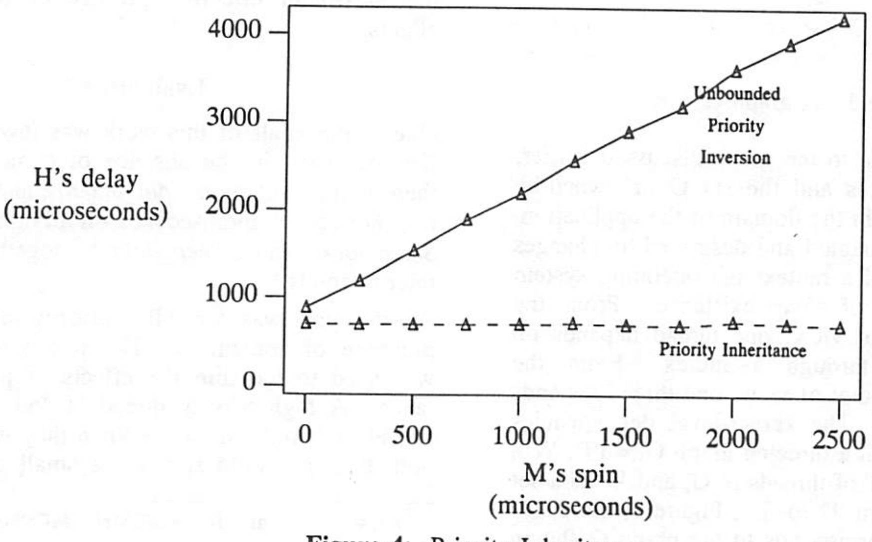


Figure 4: Priority Inheritance

References

Anderson, Thomas E. "The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors." *IEEE Transactions on Parallel and Distributed Systems.* 1(January 1990): 6-16.

Birrell, Andrew D., "An Introduction to Programming with Threads." Palo Alto: Digital Equipment Corporation Systems Research Center, 1989.

Edler, Jan, Allan Gottlieb, and Jim Lipkis, "Considerations for Massively Parallel UNIX Systems on the NYU Ultracomputer and IBM RP3." In *Proceedings of the 1986 Winter USENIX Technical Conference,* 193-210. El Cerrito: USENIX Association, 1985.

Liu, C.L. and J.W. Layland. "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment." *Journal of the ACM.* 20(January 1973): 46-61.

Luttmer, M.L.M., H. Ribbers, and P.G. Jansen, "TUMULT-X: a real-time executive." University of Twente, Department of Computer Science, int. rep. nr. INF-89-26, 1989, ISSN 0923-1714.

Reed, David P., and Rajendra K. Kanodia, "Synchronization with Eventcounts and Sequencers." *Communications of the ACM.* 22(February 1979): 115-123.

Rajkumar, Ragunathan, Lui Sha, and John P. Lehoczky, "Real-Time Synchronization Protocols for Multiprocessors." In *Proceedings of the Real-Time Systems Symposium,* 259-269. Washington, D.C.: IEEE Computer Society Press, 1988.

Sha, Lui, Ragunathan Rajkumar, and John P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization." Carnegie Mellon University, Departments of CS, ECE, and Statistics, CMU-CS-87-181, November, 1987.

Tevanian, Avadis Jr., Richard F. Rashid, David B. Golub, David L. Black, Eric Cooper, and Michael W. Young, "Mach Threads and the Unix Kernel: The Battle for Control." In *Proceedings of the 1987 Summer USENIX Technical Conference,* 185-197. Berkeley: USENIX Association, 1987.

Mark Heuser has been with Harris Computer Systems since 1981. He and others at Harris have spent the last several years adapting UNIX for real-time applications and multiprocessor platforms. His interests are in the field of operating systems. Mark has a B.S. degree in Computer Science from the University of Central Florida. His address is: 2101 West Cypress Creek Road; Fort Lauderdale, FL 33309-1892. Reach him electronically at mlheuser@ssd.csd.harris.com.

USENIX Summer Conference
June 11-15, 1990
Anaheim, California

Evolving the Vnode Interface

David S. H. Rosenthal – Sun Microsystems

ABSTRACT

The vnode interface has succeeded in supporting a wide range of file system implementations over its 6-year history. During that time it has also had to accommodate evolution in file system semantics, and in the relationship between the file system and the virtual memory system. The effects of this evolution have been less than elegant, and pressures for further evolution are mounting.

The evolution of the interface is reviewed in order to identify the problems it has caused, and a more robust revision of the interface design proposed. This design also permits new file systems to be implemented in terms of pre-existing file system implementations; it is more like the Streams interface in this respect. The current state of a prototype implementation is described and its performance characterized.

> MELENE (*lazily*). We are very well as we are. Life without a care – every want supplied by a kind and fatherly monarch, who, despot though he be, has no other thought than to make his people happy – what have we to gain by the great change that is in store for us?
>
> Gilbert & Sullivan, *Utopia, Ltd.*

Introduction

The vnode interface was developed in 1984 to abstract out file system operations in order to support multiple file system implementations, in particular NFS[9] and the Berkeley file system.[1] Over the 6 years since then it has been very successful; many versions of the UNIX kernel use it in some form, and many different file system implementations have been based on it. During that time it has also had to accommodate evolution in file system semantics, and in the relationship between the file system and the virtual memory system.[3-5, 12] The effects of this evolution have been less than elegant, and pressures for further evolution are mounting.

I review the evolution of the interface in order to identify the problems it has caused and, using experience with a SunOS 4.1-based prototype, I propose a more robust revision of the interface design. This design also permits new file systems to be implemented in terms of pre-existing file system implementations; it is more like the Streams interface[11, 16] or layered protocols in the *x*-kernel[8] in this respect. I also characterize the performance of the prototype.

It is important to stress that I am merely reporting the current state of research work in progress; I know of no current plans to make changes like the ones I describe in Sun products.

Review of Vnode Evolution

The original vnode design had the following goals:

- Provide a well-defined interface between file system implementations and the rest of the kernel.
- Support but not require Unix file system semantics. In particular it should support local disk file systems, statefull and stateless remote file systems, and other file systems such as that of MS-DOS.
- Define an interface that can be used by kernel-resident implementations of remote file servers.
- All file system operations should be atomic.

```
enum vtype {
        VNON, VREG, VDIR, VBLK,
        VCHR, VLNK, VSOCK, VBAD
};
struct vnode {
        u_short             v_flag;
        u_short             v_count;
        u_short             v_shlockc;
        u_short             v_exlockc;
        struct vfs        *v_vfsmountedhere;
        struct vnodeops             *v_op;
        union {
                struct socket  *v_Socket;
                struct stdata  *v_Stream;
        } v_s;
        struct vfs         *v_vfsp;
        enum vtype          v_type;
        caddr_t             v_data;
};
struct vnodeops {
        int      (*vn_open)();
        int      (*vn_close)();
        int      (*vn_rdwr)();
        int      (*vn_ioctl)();
        int      (*vn_select)();
        int      (*vn_getattr)();
        int      (*vn_setattr)();
        int      (*vn_access)();
```

The extract from the System V Release 4 *vnode.h* file is copyright AT&T and is reproduced by kind permission.

```
        int     (*vn_lookup)();
        int     (*vn_create)();
        int     (*vn_remove)();
        int     (*vn_link)();
        int     (*vn_rename)();
        int     (*vn_mkdir)();
        int     (*vn_rmdir)();
        int     (*vn_readdir)();
        int     (*vn_symlink)();
        int     (*vn_readlink)();
        int     (*vn_fsync)();
        int     (*vn_inactive)();
        int     (*vn_bmap)();
        int     (*vn_strategy)();
        int     (*vn_bread)();
        int     (*vn_brelse)();
};
```

Figure 1: Original Vnode Interface

To these, the implementation added:

- There should be little or no performance degradation.
- Static table sizes should not be required.
- File systems should not be forced to use central resources.
- The interface should be re-entrant.
- An "object-oriented" programming approach should be used.
- Each operation is done on behalf of the current process.

These goals resulted in the design outlined in Figure 1. Each vnode contains a small amount of data and a pointer to an "ops vector", a structure defining the operations that the rest of the kernel can invoke on the vnode object. In C++ terminology,[15] the entries in the ops vector are the virtual functions of the vnode class. The detailed semantics of the virtual functions of the various vnode versions aren't important for the argument of this paper.

These operations are invoked via macros like:

```
#define VOP_FOO(vp) (*(vp)->v_op->vn_foo)(vp)
```

Note that for a particular machine architecture, these structures and the calling conventions for the functions define a binary interface; C is used here merely as a way of making it legible.

SunOS 4.X

By the release of SunOS 4.0 in 1988, considerable evolution had occurred in the vnode interface, as shown in Figure 2. Three fields had been added to the vnode, increasing its size by 8 bytes, and 9 entries had been added to and 4 entries deleted from the ops vector (see Table 1). These changes were motivated by:

- The rewrite of the virtual memory system. This unified file I/O and paging, replacing the buffer cache operations (*vn_bmap*, *vn_strategy*, *vn_bread*, *vn_brelse*) with paging operations (*vn_getpage*, *vn_putpage*, *vn_map*), and required a new field (*v_Pages*) in the vnode.

```
enum vtype {
        VNON, VREG, VDIR, VBLK,
        VCHR, VLNK, VSOCK, VBAD, VFIFO
};
struct vnode {
        u_short                 v_flag;
        u_short                 v_count;
        u_short                 v_shlockc;
        u_short                 v_exlockc;
        struct vfs             *v_vfsmountedhere;
        struct vnodeops        *v_op;
        union {
                struct socket   *v_Socket;
                struct stdata   *v_Stream;
                struct page     *v_Pages;
        } v_s;
        struct vfs             *v_vfsp;
        enum vtype              v_type;
        dev_t                   v_rdev;
        long                   *v_filocks;
        caddr_t                 v_data;
};
struct vnodeops {
        int     (*vn_open)();
        int     (*vn_close)();
        int     (*vn_rdwr)();
        int     (*vn_ioctl)();
        int     (*vn_select)();
        int     (*vn_getattr)();
        int     (*vn_setattr)();
        int     (*vn_access)();
        int     (*vn_lookup)();
        int     (*vn_create)();
        int     (*vn_remove)();
        int     (*vn_link)();
        int     (*vn_rename)();
        int     (*vn_mkdir)();
        int     (*vn_rmdir)();
        int     (*vn_readdir)();
        int     (*vn_symlink)();
        int     (*vn_readlink)();
        int     (*vn_fsync)();
        int     (*vn_inactive)();
        int     (*vn_lockctl)();
        int     (*vn_fid)();
        int     (*vn_getpage)();
        int     (*vn_putpage)();
        int     (*vn_map)();
        int     (*vn_dump)();
        int     (*vn_cmp)();
        int     (*vn_realvp)();
        int     (*vn_cntl)();
};
```

Figure 2: SunOS 4.X Vnode Interface

- The representation of special files as a file system type. This added a field (*v_rdev*) and an operation (*vn_realvp*).
- System V support. This added a field (*v_filocks*), an operation (*vn_cntl*), and a new vnode type (VFIFO).

System V Release 4

By the release of System V Release 4 in 1989, the vnode had evolved further. As shown in Figure 3, the structure had lost 3 fields and gained 8 (all reserved for future use), expanding to 72 bytes. The ops vector had gained 8 actual operations plus another 32 reserved for future use.

```
enum vtype {
        VNON, VREG, VDIR, VBLK,
        VCHR, VLNK, VFIFO, VXNAM, VBAD
};
struct vnode {
        u_short           v_flag;
        u_short           v_count;
        struct vfs        *v_vfsmountedhere;
        struct vnodeops   *v_op;
        struct vfs        *v_vfsp;
        struct stdata     *v_stream;
        struct page       *v_pages;
        enum vtype        v_type;
        dev_t             v_rdev;
        caddr_t           v_data;
        struct filock     *v_filocks;
        long              v_filler[8];
};
struct vnodeops {
        int      (*vop_open)();
        int      (*vop_close)();
        int      (*vop_read)();
        int      (*vop_write)();
        int      (*vop_ioctl)();
        int      (*vop_setfl)();
        int      (*vop_getattr)();
        int      (*vop_setattr)();
        int      (*vop_access)();
        int      (*vop_lookup)();
        int      (*vop_create)();
        int      (*vop_remove)();
        int      (*vop_link)();
        int      (*vop_rename)();
        int      (*vop_mkdir)();
        int      (*vop_rmdir)();
        int      (*vop_readdir)();
        int      (*vop_symlink)();
        int      (*vop_readlink)();
        int      (*vop_fsync)();
        void     (*vop_inactive)();
        int      (*vop_fid)();
        void     (*vop_rwlock)();
        void     (*vop_rwunlock)();
        int      (*vop_seek)();
        int      (*vop_cmp)();
        int      (*vop_frlock)();
        int      (*vop_space)();
        int      (*vop_realvp)();
        int      (*vop_getpage)();
        int      (*vop_putpage)();
        int      (*vop_map)();
        int      (*vop_addmap)();
        int      (*vop_delmap)();
        int      (*vop_poll)();
        int      (*vop_dump)();
        int      (*vop_pathconf)();
        int      (*vop_filler[32])();
};
```

Figure 3: SVR4 Vnode Interface

Among the motivations for these changes were:

- The replacement of Unix domain sockets by Streams, which removed a vnode type and a field in the vnode.
- The need to support Xenix semantics, which added a vnode type.
- The removal of Berkeley-style locks, which removed two vnode fields.
- Additional remote file system support, which added 5 new operations.

Problems of Evolution

The evolution of the vnode is summarized in Table 1. There has been a steady growth in the size of the vnode and the number of operations in the ops vector.

Table 1: Vnode Evolution

Release	Year	Fields	Bytes	Ops
SunOS 2.0	1985	11	32	24
SunOS 4.0	1988	14	40	29
SVR4 - fill	1989	11	40	37
SVR4 + fill	1989	19	72	69
Prototype	1989	6	20	39

In System V Release 4 almost half the vnode structure and almost half the ops vector are devoted to preparing for future evolution. I believe this demonstrates that our current techniques for dealing with the evolution of kernel interfaces are inadequate. It cannot be a good idea to impose 80+% space overheads on data structures in order to cope with future change. It appears that a revision of the vnode interface to be more robust in the face of changing demands for file system functionality is required.

Design

What is this 80% overhead intended to achieve? The problem is that customer kernels have to be built from components supplied in object form by a number of suppliers. We want to let independent software vendors (ISVs) supply file system implementations if they need new file system semantics to achieve their ends, and to do so in object form to protect their investment. The processes of releasing and distributing software mean that customers cannot expect to get both a new operating system release and the corresponding release of the ISV's product at the same time.

Thus, it must be possible for the customer to build a working kernel from object components with the kernel at a higher release number than some of the file system implementations it is using. Note that there is in general no customer demand for kernels in which the file system implementations are at a higher level than the rest of the kernel; in this situation it is relatively easy for the ISV to supply both old and new versions of their file system.

In attempting to revise the vnode interface, I had two main goals:

- To make an interface that would evolve to meet new demands more gracefully by supporting *versioning*.
- To reduce the effort needed to implement new file system functionality by allowing vnodes to be *stacked*.

The idea of stacking vnodes is not new; several of the file systems in SunOS 4.1 (the *translucent*[7] and the *loopback* file systems, for example) implement vnodes whose operators simply invoke the operators of an underlying vnode with slightly altered arguments. But there had always been severe restrictions on the ways in which vnode operations could be overridden. I wanted to be able to override all vnode operations in a completely general way that would support breaking file system functionality down into small modules like Streams modules. Instead of viewing file systems as large monolithic structures, I wanted to be able to plug them together from smaller pieces.

Design Guidelines

To investigate solutions to these problems I evolved a prototype, starting with an alpha version of SunOS 4.1. The prototype has been running for quite some time, but it is very much the result of evolution not design. As I experimented with it I evolved a set of design guidelines:

- *Vnodes should stack.* In other words, it should be possible to interpose new functionality on all operations invoked on an existing vnode without locating all pointers to it and updating them. This is especially important since each page structure contains a vnode pointer, and requiring file system code to find and update all the page structures leads to an inadmissible mixing of the file and virtual memory sub-systems.
- *In fact, vnodes should tree.* It should be possible for one higher-level vnode to represent a number of lower-level vnodes. As an example, consider a fan-out-fs whose operations simply invoke the corresponding operation on all of a set of underlying vnodes (Figure 4).

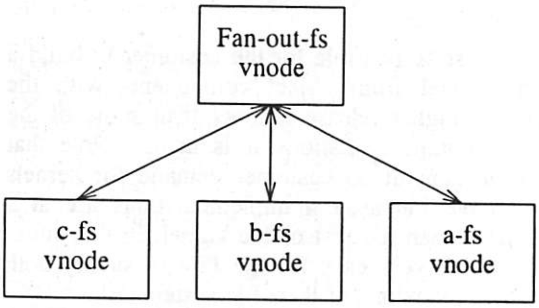


Figure 4: A vnode tree

- *No special-case code for mount.* The whole concept of stacking vnodes is a generalization of the concept of mount. It should be possible to replace the special-case code in name lookup that currently implements mounts, and the *vfs_mountedhere* field in the vnode.
- *Vnodes should be opaque.* In other words, the data structure visible to higher levels of the kernel should contain no data, only a pointer to the ops vector. A visible datum in the vnode represents a possible operation (updating the value of the datum) that cannot be overridden via the ops vector, because it doesn't need to go via the ops vector.

  Further, one of the problems of evolving the interface is that the visible data in the vnode structure changes. If there is no data in the vnode, it cannot change.
- *Vnodes should be cheap.* If vnodes are to stack (or tree) there are likely to be several vnodes where at present there is only one. So, space in the vnode is at a premium. An important advantage of opaque vnodes is that the data that they don't contain doesn't take up space. Nor is there any need to reserve space in the shape of *v_filler[8]* in case the non-existent data expands.

  It might be argued that all that opaque vnodes achieve is to move data from the public to the private part of a vnode. But, to my surprise, much of the data in the public part of the SunOS 4.X vnode was often present in some other form in the private part.
- *No vnode type.* A vnode type field or a vnode type operator is an invitation for the higher-level kernel code to test the type and behave differently depending on the result. This is likely to cause difficulties for file systems trying to intercept all operations. It is better for the rest of the kernel to treat all vnodes equally and leave all special case catching and error generation to the file-system specific code.

  For example, the SunOS 4.X kernel checks to see if the vnode in which it is being asked to look up a name is a directory, and generates ENOTDIR if it isn't. Why not simply call the vnode's *vn_lookup()* operator and let it generate the ENOTDIR if the vnode doesn't represent something that can have names looked up in it?
- *There should be a cheap way to lock the whole stack of vnodes.* While one process is manipulating a stack of vnodes, other processes must be prevented from invoking operations on any of the vnodes in the stack. Locking the stack in this sense shouldn't involve, for example, traversing the stack setting a lock flag in each vnode.
- *Vnodes should support versioning.* It should be possible for the higher-level kernel code to use file system implementations constructed with several versions of the interface, and similarly to build file system implementations that work with kernels that implement several different versions of the interface.

The New Vnode

I believe that the new vnode should look like Figure 5. This isn't quite the way it currently looks in my prototype*, but the prototype bears the scars of a lot of exploratory hacking.

```
struct vnode {
                struct vnode    *v_top;
           struct vnode      *v_above;
           struct vnodeops *v_op;
           u_short           v_readers;
           u_short           v_count;
           caddr_t           v_data;
};
struct vnodeops {
           int       (*vn_version)();
           int       (*vn_open)();
           int       (*vn_close)();
           int       (*vn_rdwr)();
           int       (*vn_ioctl)();
           int       (*vn_select)();
           int       (*vn_getattr)();
           int       (*vn_setattr)();
           int       (*vn_access)();
           int       (*vn_lookup)();
           int       (*vn_create)();
           int       (*vn_remove)();
           int       (*vn_link)();
           int       (*vn_rename)();
           int       (*vn_mkdir)();
           int       (*vn_rmdir)();
           int       (*vn_readdir)();
           int       (*vn_symlink)();
           int       (*vn_readlink)();
           int       (*vn_fsync)();
           int       (*vn_inactive)();
           int       (*vn_lockctl)();
           int       (*vn_fid)();
           int       (*vn_getpage)();
           int       (*vn_putpage)();
           int       (*vn_map)();
           int       (*vn_dump)();
           int       (*vn_cmp)();
           int       (*vn_cntl)();
           int       (*vn_vfsp)();
           int       (*vn_socket)();
           int       (*vn_stream)();
           int       (*vn_bsdlock)();
           int       (*vn_bsdunlock)();
           int       (*vn_isswap)();
           int       (*vn_swapon)();
           int       (*vn_push)();
           int       (*vn_pop)();
           int       (*vn_namepage)();
           int       (*vn_unnamepage)();
           int       (*vn_pagecount)();
           int       (*vn_pageapply)();
};
```

Figure 5: The Ideal Vnode

Note that the vnode is almost opaque; except for the reference count all the visible fields are needed to implement the vnode stack. This is held together by the *v_top* and *v_above* fields as shown in Figure 6.

There is no public *v_below* pointer. The representation of the set of vnodes, if any, below a vnode is private to that vnode's file system implementation; no higher-level code knows or cares about the vnodes that may exist below a vnode. The absence of a public down pointer is a principle; if there were one it would allow higher-level code to traverse the vnode trees and invoke lower-level vnode's operations without the intervening vnode finding *out.*

The higher-level code invokes operations using its vnode pointer like this:

```
#define VOP_FOO(vp) (*(vp)->v_top->v_op->vn_foo)(vp)
```

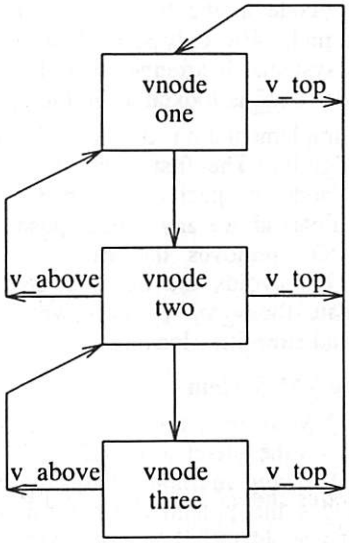


Figure 6: Vnode Links

Instead of indirecting through the vnode to find the ops vector and invoking the appropriate operation as with previous vnode interfaces, the macro finds the top vnode of the stack and invokes the appropriate operation from *its* ops vector. Even if the vnode pointer points into the middle of the stack, the code that gets invoked will be the corresponding operation of the top vnode of the stack. Whether that vnode executes the operation itself, or forwards it to other vnodes below it, is no concern of the invoker's. In effect, a pointer to a vnode becomes an alias for the top vnode of the stack.

Reference counting of these links is simple; the *v_above* pointers hold a counted reference to the vnode they point to but the *v_top* pointers do not. Normally, the file-system specific downward links do hold their vnode.

Implementation

Stack Manipulation Operators

Vnode stacks can be manipulated by two new vnode operators, VOP_PUSH and VOP_POP.

```
VOP_PUSH(below, above)
```

*The prototype hasn't yet demonstrated versioning, some of its vnode operations return values other than an error code, and its ops vector still contains some operations that should be obsoleted. Further, it isn't compatible with SunOS 4.1 in that it returns the wrong code on some errors.

arranges that *above* is the new top vnode of the stack containing *below*. If *vp* is the top of a stack,

```
VOP_POP(vp)
```

pops it off. The implementation of these operators is file-system specific because they must deal with the private representation each implementation uses for the set of vnodes below a vnode, and because these operations may have file-system specific side effects.

The mount operations use these facilities. A mount is simply a stack of a root directory over a mount point directory; lookup operations on the lower vnode automatically look up in the upper vnode with no special-case code in the file-system independent parts of the kernel. The only special-case code left is in each file system. It arranges to fall through to the lower vnode if ".." is looked up in the upper one.

In the current implementation, the definition of "top" is stretched slightly. The first time VOP_PUSH is applied to a vnode, a special stack head vnode is created which floats above any vnodes pushed on the stack. VOP_POP removes the vnode below this stack head. This avoids the need to descend the stack and update the *v_top* pointers when pushing and popping, and simplifies locking.

Stacks and the VM System

In the SunOS VM system, the page cache uses the vnode pointer and the offset to establish the identity of the page. Allowing multiple aliases for a vnode pointer introduces the potential for aliasing in the page cache. To avoid this, it is necessary to establish a rule for file system implementations.

The rule is, if a vnode belonging to a file system implementation that wants control over the page cache is pushed onto a stack, it must claim all existing pages for the stack below. When it is popped, it must either ensure there are none of its pages in the cache or restore the previous identity.

A page in the cache is labelled with its identity by the VOP_NAMEPAGE operator, and unlabelled by VOP_UNNAMEPAGE. With the current VM system, these operators add and delete pages from the page hash lists. A function, such as one that renames pages, can be applied to all the cached pages for a vnode by:

```
error = VOP_PAGEAPPLY(vp, fn, data);
```

In principle, this interface allows each file system implementation to choose its own representation for the set of in-memory pages. In practice, more work is needed to refine the VM/FS interface to make this feasible. The VM cache and locking (see the next section) have an incestuous relationship, and I'm far from happy with the current implementation in either respect.

Locking

In order to ensure other processes don't see a malformed vnode stack, there must be a method of locking the entire stack while it is being manipulated. It must be cheap and cannot involve traversing the stack itself.

The technique I'm currently working on puts a readers/writers lock in the special stack head vnode. Before changing the stack, a writer switches the ops vector in the stack head from one containing operations like Figure 7(a) to one containing operations like Figure 7(b). These then "capture" any process invoking an operation on any vnode pointer in the stack and puts it to sleep.

```
static int
snarefs_foo(vp)
    struct vnode *vp;
{
    return(VOP_FOO(vp));
}
```

Figure 7(a): Vnode Operation When Unlocked

```
static int
snarefs_foo(vp)
    struct vnode *vp;
{
    int error;
    int s;

    s = splhigh();
    (void) VOP_HOLD(vp);
    while (vp->v_top->v_op == &snarefs_slow_ops) {
        (void) sleep((caddr_t)
            &(vp->v_top->v_op), PSNARE);
    }
    (void) splx(s);
    error = VOP_FOO(vp);
    (void) VOP_RELE(vp);
    return (error);
}
```

Figure 7(b): Vnode Operation When Locked

The writer waits until there are no readers, edits the stack, and switches the ops vector of the stack head back. This technique has no overhead for vnodes that aren't part of a stack, and only a tail-recursion-type procedure call for stack vnodes that are not locked. However, the cost of maintaining the count of the number of readers in a stack is significant.

Versioning

As described above, one advantage of opaque vnodes is that there is no need to version the vnode structure itself. But the evolution so far indicates that change in the ops vector must be anticipated. Fortunately, stacking opaque vnodes allows an "adaptor file system" to be defined to convert from the new version of the interface used by the kernel to the older interface used by the file system implementation (see Figure 8). All that the adaptor-fs vnode needs in its private data is the down pointer.

This technique's overheads are very small:

- no increase in size of the vnode structure,
- no increase in size of the ops vector,

- a small increase in run time for those vnodes supported by back-level file system implementations.

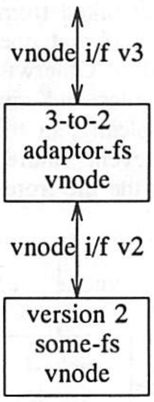


Figure 8: Adaptor-fs - v3 kernel to v2 fs

- a small increase in space consumption, for the adaptor-fs vnode for each vnode supported by a back-level file system implementation.

Despite this, the technique allows for more change in the interface than adding fillers. There are no arbitrary limits on how much the interface can change, provided the adaptor-fs can emulate the minimum required new functionality in the old interface's terms.

Performance

In my work on the prototype I have been exploring rather than tuning, but I have found one performance technique that could also be applied to the existing vnode interfaces. Current file system implementations have a single vnode ops vector; all vnodes they create have the same ops vector pointer (in fact, unpleasant parts of some kernels even use the ops vector pointer to decide which file system the vnode comes from!).

I changed the file systems to use a separate ops vector for each type of vnode they support. For example, the UFS file system implementation has a different vector for normal files and for directories. The *vn_lookup* entry in the directory vector points to *ufs_lookup*, but in the file vector it points to a routine like:

```
int
vfs_enotdir(vp)
        struct vnode *vp;
{
        return (ENOTDIR);
}
```

In this way, the file system implementation doesn't have to start all its operations by examining the type field and generating errors if it's wrong. This computation is done once at vnode create time instead of every time an operation is invoked. In fact, the prototype contains a vnode ops vector supported by the "Nancy Reagan" file system – *all* its ops vector entries point to error stubs like *vfs_enotdir()*.

In this framework, just as each vnode type can have its own ops vector, it can also have its own private data. When defining the inode-equivalent for a new file system, there is no need to provide space for the directory offset in the private data for a regular file.

Running one of Sun's synthetic workload benchmark suites on the prototype kernel initially revealed approximately 7% degradation, with special benchmarks for name lookup showing approximately 30% degradation. This was obviously unacceptable, even for a prototype. The major causes of degradation turned out to be:

- The vnode reference counting; in my zeal for completely opaque vnodes I had turned the VN_HOLD and VN_RELE macros into full-blown vnode operations called through the ops vector (8%).
- Detecting vnodes that are the root of a mounted file system; I had eliminated the VROOT flag in a very expensive way (18%).

Fixing these resulted in a kernel with no detectable degradation on the synthetic workload benchmarks and about 1% degradation in name lookup. The worst-case path I found was looking up ".." across a mount point, with about 6% degradation. However, this was all before I started working on locking. The various locking implementations I've tried so far have degradations in the 15-25% range for the worst-case paths in name lookup.

Using File System Modules

As well as providing for evolution this way of decomposing file system functionality into modules and connecting them at run-time in various ways allows many opportunities for new ways of implementing file system functionality. To illustrate them, I describe a few possible modules. I haven't implemented these ideas yet, I'm only discussing them to show the potential of the concept.

Quotas

Support for file-space quotas is an interesting example of the possible use of file system modules. In the current SunOS source, *#ifdef QUOTA* appears in 22 files, ranging from *machdep.c* to *init_main.c*. Despite this, only the *ufs* file system supports quotas.

Suppose we construct a file system module that can be pushed on top of mounted file systems to provide quota services. It would intercept all operations to the file systems underneath and maintain an internal space usage database. Operations violating the quotas would be rejected. The quota-fs would use normal file system operations on the underlying file system to externalize the space database.

In this way, a single file system module implementation with no *#ifdef QUOTA* elsewhere in the kernel could provide quotas for *all* other file system implementations.

A Less Temporary File System

One of the performance problems with *ufs* is the need to write directories synchronously in order to ensure that they will still be there in a consistent form if the system crashes. This is a particular problem in */tmp*, where the user may not care if the files survive a crash. SunOS includes a file system implementation called *tmpfs* which represents files in virtual memory; they may get paged out to the swap area but they are never written to disk. Mounting this on */tmp* and */usr/tmp* improves performance significantly, at the cost of ensuring that *no* files in */tmp* and */usr/tmp* survive a crash.

There is only one problem with this approach. Some applications, *vi* for example, require temporary files to survive crashes and be scavenged before */tmp* is cleaned out. Fortunately, these applications normally *fsync()* their temporary files when checkpointing to make sure they don't hang around in the buffer cache and get caught by a crash. We can exploit this by building a write-on-fsync module like Figure 9.

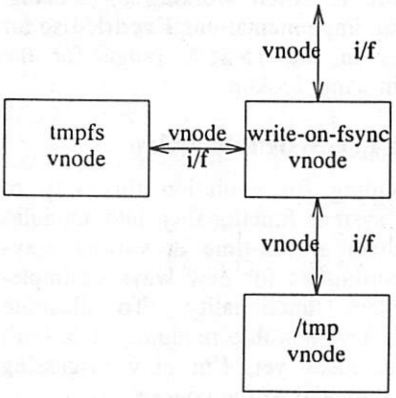


Figure 9: Less Temporary File System

The module would route all writes to the *tmpfs* vnode until and unless the process invoked the *fsync()* operator. At that point, the file would be copied to the underlying file system. In this way, files that were never *fsync()*ed would get the full benefit of being really temporary, and files that were would actually appear in permanent storage.

Watchdogs

Another possible module would implement *Watchdogs* as described by Bershad and Pinkerton.[2] The watchdog-fs vnode would intercept all or only the selected operations, convert them into IPC messages on a Stream, and wait for a response from the user-level watchdog process reading the Stream. The content of this response would determine if the operation was to be passed on down the stack, or returned to the invoker.

Read-Only Caching

Consider a module like that in Figure 10. When one of its operations is invoked from above, it forwards it to the a-fs vnode. If that operation succeeds the result is returned. Otherwise, the operation represents a cache miss and must be forwarded to the b-fs vnode. Typically, in this case the cache-fs will also try to prevent future cache misses, for example by copying the file from the b-fs to the a-fs.

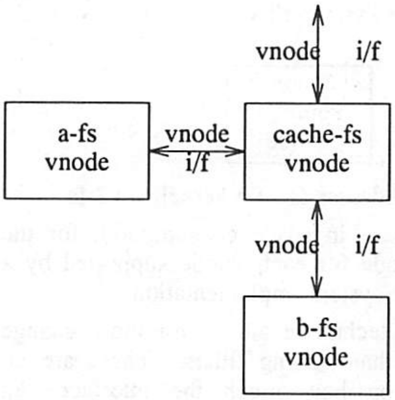


Figure 10: RO Cache File System

This simple module can use any file system as a file-level cache for any other (read-only) file system. It has no knowledge of the file systems it is using; it sees them only via their opaque vnodes. Figure 11 shows it using a local writable ufs file system to cache a remote read-only NFS file system, thereby reducing the load on the server. Another possible configuration would be to use a local writable ufs file system to cache a CD-ROM, obscuring the speed penalty of CD.

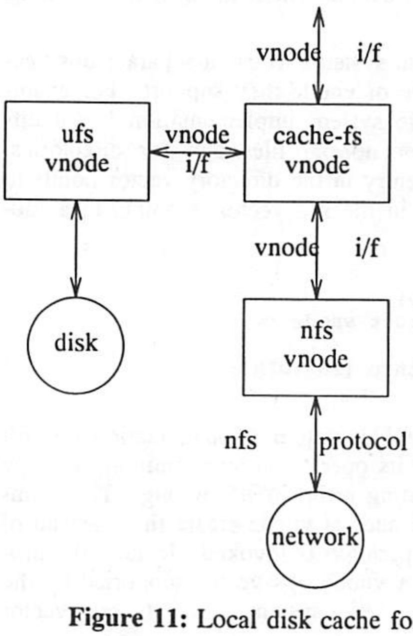


Figure 11: Local disk cache for RO NFS

File-level caching can be inefficient. If the file system being used as a cache can be persuaded, for example via a flag on the VOP_RDWR call, to reject reads to holes in files instead of returning zeros it can be used as a block-level cache. A new entry in the cache is created as a file full of holes, which are filled in as reads to the a-fs vnode fail with an appropriate error such as EHOLE. This is a very simple change to UFS.

Read-Write Caching

Srinivasan and Mogul[13,14] modified the NFS protocols by adding the cache-consistency protocols from the Sprite operating system.[10] In this environment, adding cache-consistency and thus enabling a read-write file cache need not involve modifying the NFS protocol. By allowing the cache-fs implementations to communicate amongst themselves using a cache-consistency protocol such as Sprite's or the V system's *leases*[6] alongside the NFS protocol we should be able to add caching without changing the protocol used to move data to and fro (see Figure 12).

Fall-back

Another useful module could be the fall-back-fs, shown in Figure 13. Each operation it receives from above is sent to one of the N underlying vnodes. If it fails with a retryable error, or does not return before a timeout, another of the vnodes is chosen and the operation tried on it. In this way, the reliability and availability of the file system seen through the top vnode interface is greater than any of the individual underlying file systems, and this has been accomplished with no knowledge of or modification to the underlying file systems.

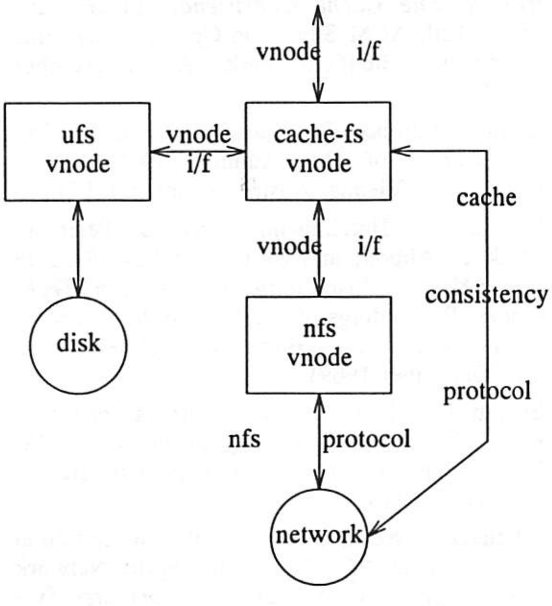


Figure 12: Local disk cache for RW NFS

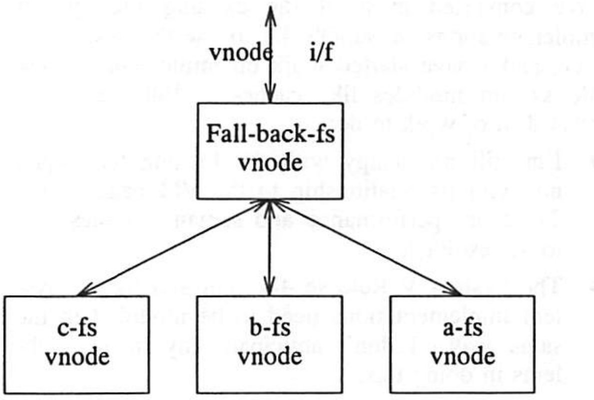


Figure 13: Fall-back-fs

Typically, this would be used to spread the load among a number of NFS file servers for read-only file systems, and to avoid clients being blocked when one of the servers went down.

Replication

A similar useful module is the replicate-fs, shown in Figure 14. Each operation it receives from above is sent to *each* of the N underlying vnodes. When M (<=N) of them has returned successfully, replicate-fs returns upwards. Read operations choose one of the N randomly. Implementations of replicate-fs communicate with each other using an ordering protocol to ensure that each sees the same sequence of operations and therefore stays consistent.

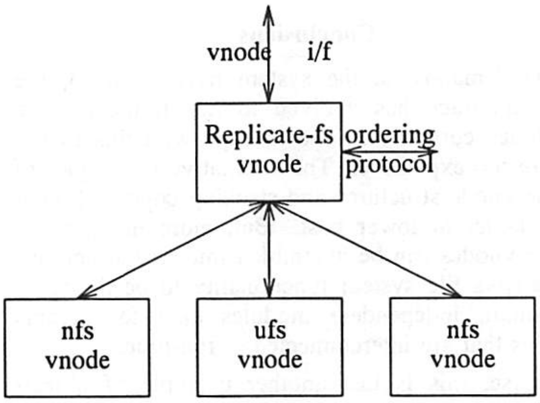


Figure 14: Replicate-fs

Using this module, a reliable replicated configuration could be assembled from unreliable pieces. For example, a set of machines could each operate by replicating operations across a local copy of a file system and a remote mount of each of the other system's copies.

Future Work

I have only started exploring these ideas. I have a working prototype SunOS 4.1 kernel that is close to the vnode interface I prefer, but I have done little work on the versioning aspects of the problem. I have converted most of the existing file system implementations in SunOS 4.1 to use the new interface, and I have started work on implementing new file system modules like cache-fs. But there is a great deal of work to do:

- I'm still not happy with the locking technique, nor with its relationship to the VM page cache. There are performance and semantic issues still to be resolved.
- The System V Release 4 kernel and its file system implementations need to be modified in the same way. I don't anticipate any major problems in doing this.
- More file system modules need to be implemented.
- The adaptor-fs versioning technique needs to be implemented and tested.
- Experiments using the same techniques on other internal kernel interfaces, particularly the VFS interface, are needed.
- The whole question of multi-threading has been totally ignored.

Assuming that these are all achieved, there are a large number of questions to be answered about what to do with this technology. The vnode interface is part of the System V Release 4 standard, and is thus under formal change control. Discussions will be needed to investigate what role these techniques might play in the future of System V.

Conclusions

As the demands on the system have changed, the vnode interface has evolved to match them. The techniques currently in use to cope with this evolution are too expensive. The alternative techniques of opaque vnode structures and stacking cope with evolution better at lower cost. But, more importantly, opaque vnodes can be assembled into tree structures. This allows file system functionality to be dissected into small, independent modules akin to Streams modules that are interconnected at run-time.

Of course, this is just another example of object-oriented programming. But unlike most examples, this is object-oriented programming at the binary level.

Acknowledgements

Many of these ideas have been rattling around in the back of my mind since the early discussions about the Andrew File System at Carnegie-Mellon's Information Technology Center, and they owe much to Bob Sidebotham. In particular, a somewhat hysterical brainstorming session at an ITC party between Bob, David Nichols and myself led to the ''holey cache-file'' concept.

Steve Kleiman and Bill Joy designed the original vnode interface, and have been extraordinarily helpful in my efforts to change it. The same applies to all my colleagues in the Systems Group at Sun, especially to Bill Shannon, Rob Gingell, Mike Powell, and Glenn Skinner. Special thanks are due to Steve Baumel for penetrating reviews of drafts of the paper. I'm also very grateful to Sun's management for their patience; although this project has already taken many times longer than my initial estimate, and it is still far from finished, they have always encouraged me to take the time to ''do the right thing''.

References

1. M. Kirk McKusick *et al*, ''A Fast File System for UNIX,'' *ACM TOCS* **2**(3) pp. 181-197 (August 1984).
2. Brian N. Bershad and C. Brian Pinkerton, *Watchdogs: Extending the UNIX File System,* Proceedings of the Winter 1988 Usenix Conference, Dallas, TX (February 1988).
3. Howard Chartock, *RFS in SunOS,* Proceedings of Summer Usenix Conference, Phoenix, AZ (June 1987).
4. Robert A. Gingell, Joseph P. Moran, and William A. Shannon, *Virtual Memory Architecture in SunOS,* Proceedings of Summer Usenix Conference, Phoenix, AZ (June 1987).
5. Ed Gould, *The Network File System Implemented on 4.3BSD,* Proceedings of Summer Usenix Conference, Atlanta, GA (June 1986).
6. Cary G. Gray and David R. Cheriton, *Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency,* Proceedings of the 12th ACM Symp. on Operating Systems Principles, Litchfield Park, AZ (December 1989).
7. David Hendricks, *The Translucent File Service,* Proceedings of the Autumn 1988 EUUG Conference, Vienna, Austria (October 1988).
8. Norman C. Hutchinson, Larry L. Peterson, Mark B. Abbott, and Sean O'Malley, *RPC in the x-Kernel: Evaluating New Design Techniques,* Proceedings of the 12th ACM Symp. on Operating Systems Principles, Litchfield Park, AZ (December 1989).
9. Steven R. Kleiman, *Vnodes: An Architecture for Multiple File System Types in Sun UNIX,* Proceedings of Summer Usenix Conference, Atlanta, GA (1986).
10. Michael N. Nelson, Brent B. Welch, and John K. Ousterhout, ''Caching in the Sprite Network File System,'' *ACM Trans. on Computer Systems* **6**(1) pp. 134-154 (February 1988).
11. Dennis M. Ritchie, ''A Stream Input-Output

System,'' *AT&T Bell Laboratories Tech. J.* **63**(8)(October 1984).

12. Mordecai B. Rosen and Michael J. Wilde, *NFS Portability,* Proceedings of Summer Usenix Conference, Atlanta, GA (June 1986).
13. V. Srinivasan and Jeffery C. Mogul, ''Spritely NFS: Implementation and Performance of Cache-Consistency Protocols,'' Research Rept. 89/5, Digital Western Research Lab., Palo Alto, CA (May 1989).
14. V. Srinivasan and Jeffery C. Mogul, *Spritely NFS: Experiments with Cache-Consistency Protocols,* Proceedings of the 12th ACM Symp. on Operating Systems Principles, Litchfield Park, AZ (December 1989).
15. Bjarne Stroustrup, *The C++ Programming Language,* Addison-Wesley, Reading, MA (1987).
16. Ian Vessey and Glenn Skinner, *Implementing Berkeley Sockets in System V Release 4,* Proceedings of the Winter 1990 USENIX Conference, Washington, DC (January, 1990).

David Rosenthal is a Distinguished Engineer at Sun Microsystems. He holds a M. A. degree from Cambridge University and a Ph. D. from the University of London, and has worked on computer graphics, user interface technologies, and operating systems at the University of Edinburgh's Architecture Dept., the CWI Amsterdam, and Carnegie-Mellon University's Information Technology Center. He looks on this paper as a sign that he is making progress in recovering from a seven-year addiction to window systems.

USENIX Summer Conference
June 11-15, 1990
Anaheim, California

Bruce Thompson, Daryl Stolte, David Ellis
– Hewlett-Packard Company

A Transparent Integration Approach for Rewritable Optical Autochangers

ABSTRACT

Insufficient disk space has been a problem with computer systems since their introduction. The high cost of online peripherals and limited floor space constrains the amount of disk storage a system user can afford. Optical storage, in conjunction with autochangers promises to relieve the online data crunch by providing storage at a cost competitive with tapes but with access speeds approaching disks. Because these Direct Access Secondary Storage (DASS tm) devices have unique attributes, a new method of peripheral integration is required.

This paper describes an integration technique for re-writable optical storage and autochangers for the UNIX operating system that is transparent to users and applications. All the standard UNIX methods of accessing and managing files work on the autochanger as they do with disks. No new commands or utilities are needed.

Introduction

Rewritable optical technology has become a viable storage technology. Optical drives resemble normal winchester drives in their general behavior and capacity. However, the durable design of the optical disk allows for safe removal from the optical drive[1]. This makes an optical disk autochanger possible.

The autochanger can be used to allow access to huge amounts of data in a single unit without the need for operators or users to perform the exchanging of the cartridges[2]. Since the capacity of each cartridge is around 650 megabytes (325 on each side) [1], an autochanger with a capacity of around 20 gigabytes in the size of a two-drawer filing cabinet is feasible.

The rewritable optical autochanger is unique in that if a request is made to one of the cartridges, a delay of a few seconds occurs while the cartridge is retrieved and inserted into a drive. Once the cartridge is in the drive, the user experiences similar performance to winchesters. Because of this behavior, it isn't really a disk drive in the sense that it is always online, and it isn't a tape in that it is not a serial device. This device falls somewhere in between the two.

Direct Access Secondary Storage

Figure 1 displays how DASS relates to traditional storage devices. Notice that there's a large gap between primary and secondary storage in terms of access time and cost per megabyte. Average access time for hard disks is measured in milliseconds. Access time for information on tapes is measured in tens of seconds for mounted tape, minutes to hours for a tape in a library. The cost of magnetic disk storage is about $10 to $20/MB, while tape storage costs around 15 to 20 cents per megabyte[3]. The advancement in disk and tape technology continues to widen this gap.

Rewritable Optical fills this gap. With an average access time in the .1 to 10 second range, and a cost of 20 to 40 cents per megabyte, optical drives creates a new segment in the storage hierarchy. However, the rewritable optical disk and autochanger technology only provide the "secondary storage" aspect of DASS. This paper discusses how the "Direct Access" is achieved.

Transparent Integration

The major goal in bringing this technology to the end user is to make its operation intuitive and convenient. The unique attributes of the device should be hidden as much as possible, traditional commands should operate identically, and

[1] The magnetic material is embedded between two sheets of plastic rather than being exposed on the surface as with winchester media. See [1] for more information.

[2] An electro-mechanical arm, guided by the host computer, selects, moves, rotates and inserts optical disk cartridges into drives.

[3] Typical for 1/2" tape. Newer tape technology drops it to about 1 cent.

application programs should not need to be modified to understand how to store and retrieve files. A device with the capacity of an autochanger (in the 10's of gigabytes) will most likely be used on a network. Therefore, all users of the network should also be able to access this device remotely. This is the essence of transparency. Transparency makes a rewritable optical autochanger a **Direct Access** Secondary Storage device.

Alternatives: The User's View of the Autochanger

There are several ways the user can view the autochanger.

- It can be treated as a disk drive(s) and a changer mechanism. Most autochangers are electrically constructed with the changer separate from the drive interfaces so at first it appears straightforward to let the user view it in the same way. However, with this model, each application would have to understand how to change cartridges. Thus, all applications would have to be modified to take advantage of this device.

- It could be viewed as one huge disk. This could be implemented as a simple driver that would perform sector address calculations to select the appropriate cartridge. However, all the cartridges must be kept together and in the correct order. Also, the capacity can easily exceed 2^32 bytes, the maximum value of the four-byte integer used in the unix seek(2) system call. Sharing individual cartridges from the autochanger with a stand-alone drive would also be difficult.

- Each cartridge surface could be viewed as though it had its own drive. When a request is made to this "pseudo" drive, the cartridge is inserted into a real drive before the request is processed.

The Implementation: One File System per Surface

We chose to integrate the autochanger using the third alternative, treating each surface like a hard disk. This alternative offers the most transparancy and flexibility. Each surface can do the following:

- Hold a file system.
- Be stored to and rewritten like a hard disk.
- Has a device file associated with it.
- Be accessed with the same unix commands as a magnetic disk.
- Be removed from the autochanger and accessed with a stand-alone drive.

To see how a typical user would interact with a system with an autochanger configured, reference the directory structure in figure 2.

In figure 2 the boxes represent file systems residing on a cartridge surface in the autochanger. These surfaces can be mounted anywhere in the entire directory structure of the system. Even though all the surfaces are mounted, which means access to those file systems can be made, not every cartridge is in a drive.

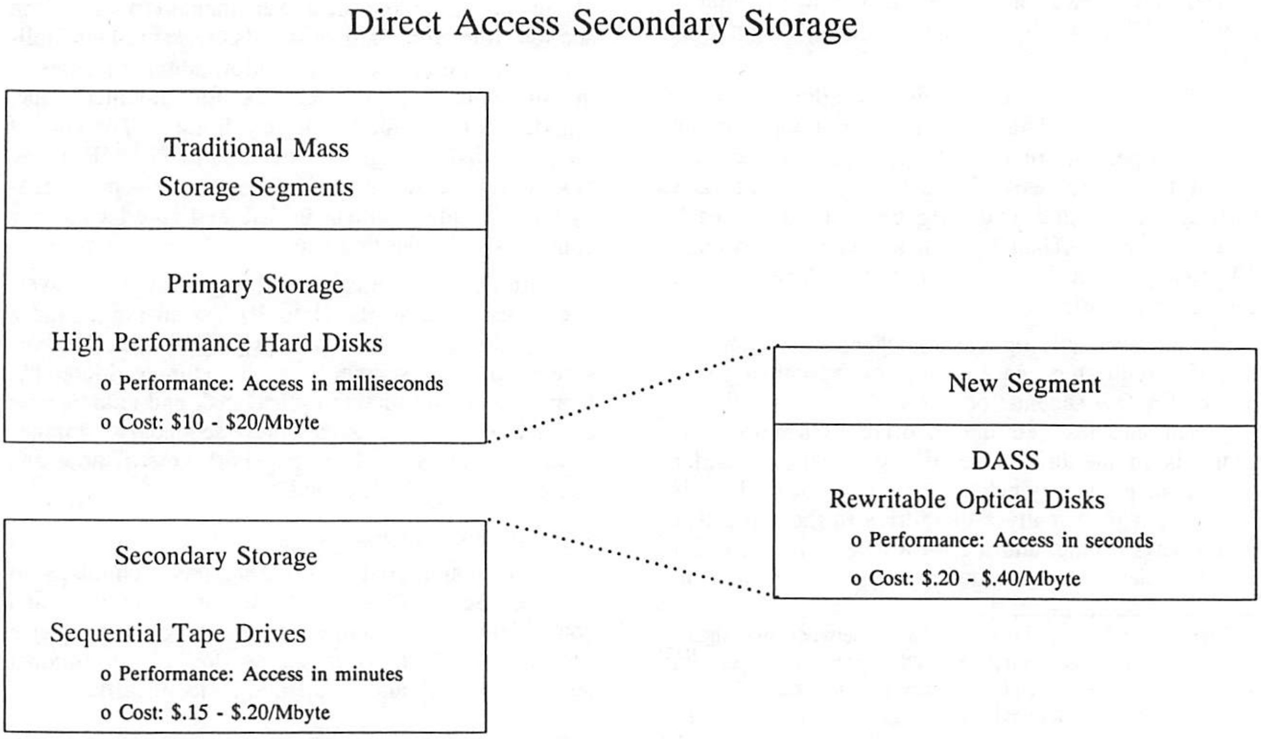

Direct Access Secondary Storage
Traditional Mass Storage Segments
Primary Storage
High Performance Hard Disks
o Performance: Access in milliseconds
o Cost: $10 - $20/Mbyte
Secondary Storage
Sequential Tape Drives
o Performance: Access in minutes
o Cost: $.15 - $.20/Mbyte
New Segment
DASS
Rewritable Optical Disks
o Performance: Access in seconds
o Cost: $.20 - $.40/Mbyte

Figure 1

The Power of Transparent Integration

The list below describes what opportunities transparent access provides. The list is certainly not exhaustive but, it does represent typical uses of the device.

- For the system administrator, each optical surface has a device special file associated with it. A file system is mounted the same as any other hard disk, using the **mount** command. For example, the following command will mount surface "1" side "a" at the directory projectx:

```
# mount /dev/ac/1a /users/davee/projectx
```

Notice the transparency, no new procedures to learn, no new commands to execute.

- Suppose, user *davee* stores information concerning an old project (projectx) on a surface in the autochanger. Since it is an old project, it is not referenced frequently but is needed when questions arise. When he does an **ls** of /users/davee/projectx, that cartridge will be put in to a drive before the request is served. The user experiences a delay in command execution until the surface is inserted into a drive. But notice that the command behaves as though it were to a hard disk. His secondary storage is being accessed directly, not consuming expensive on-line storage.

- To free hard disk space, users do not need to delete files. They simply migrate them to the direct access autochanger storage. For example, the on-line manual pages beyond section 1 tend to be infrequently accessed. Section 2 could be stored (preformatted) in the autochanger. In some cases it is faster to retrieve a preformatted copy of a document from a DASS device, than to store the unformatted version on disk and require the text formatter to run before it can be viewed.

- Another typical case is when a system administrator needs some temporary disk space to perform some function. With a DASS device, he can copy some data from disk to a cartridge in the autochanger such as /tmp/big. The data can then be copied back at a later time.

- Backing up a system and recovering a file from that backup has always been a labor intensive task. With the advent of higher capacity tape drives, this is being eased somewhat. However, with a DASS device for backup, the file system on the winchesters can be recreated in the backup. Not only the files but the file structure can be copied. The cartridge can then be mounted (probably read-only) back into the original file system. This allows users to browse files in the backup at will. Backed up files can be recovered with a simple **cp** command. Backup can be thought of as an insurance policy. A premium is paid every night to this policy. Traditionally, making a claim (recovering a file from the backup), usually requires knowing which tape the file is on, what the exact file name is, how to run the recover program, and where to put the file on hard disk once it is located. Using a DASS device, making a claim is transparent.

Figure 2 also shows how incremental backups can be remounted into the file system. If say, user *brucet* wants to access a file from Monday's backup, all he has to do is to cd to /backup/monday. Once the file is located, it does not necessarily have to be

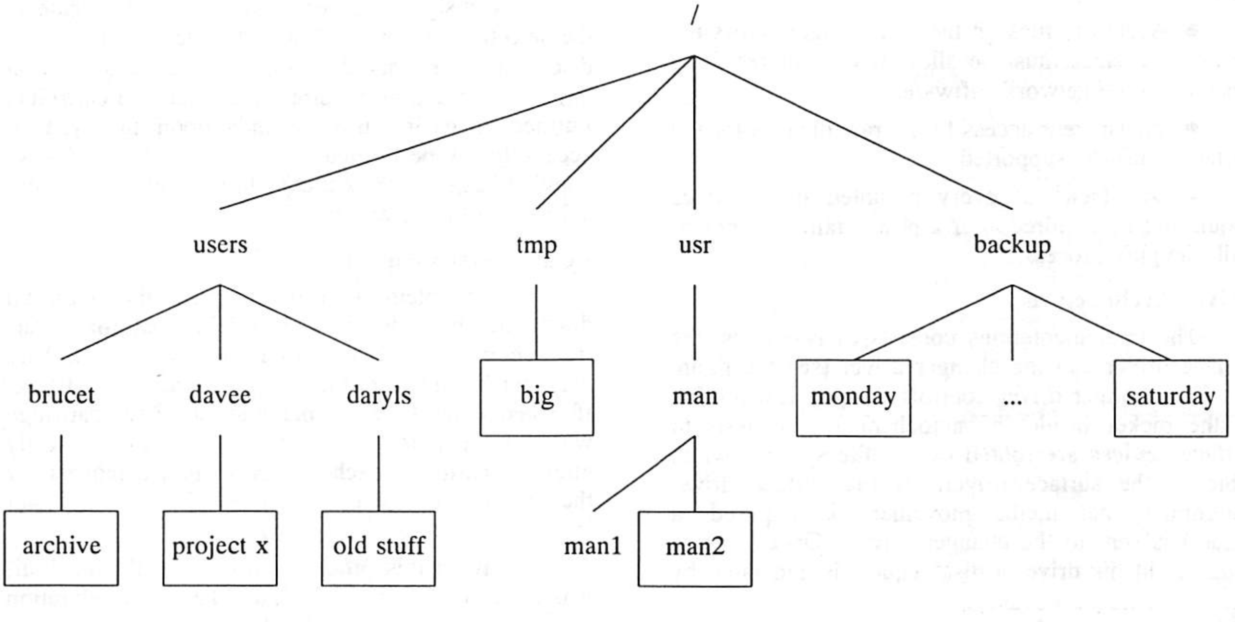


Figure 2

copied back onto disk before it is accessed. It can be viewed, executed, etc. directly from the optical disk.

- Network users benefit from transparency by being able to access the autochanger from their own workstation. If another user had /users/daryls/oldstuff mounted via NFS, he could access the files on this cartridge remotely.

- Application programs do not require special knowledge about the autochanger. A CAD/CAM program can store drawing files, part files, etc. on the autochanger by just continuing to make file system calls but with a file path that eventually ends up on an autochanger surface.

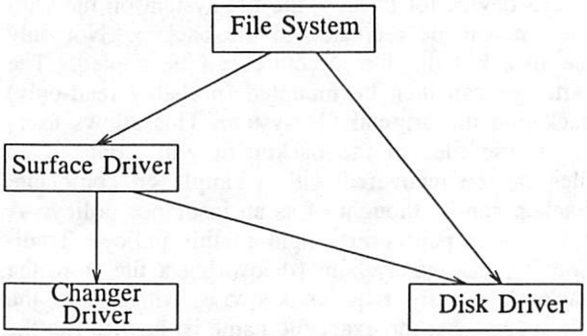


Figure 3

DESIGN

The design goals which were considered crucial to the project success were,

- Support transparent access - no special programs or utilities should be required

- Existing application programs should not have to be changed to take advantage of the autochanger.

- Access to files on the autochanger across the network services must be allowed without requiring changes to the network software.

- Transparent access to the raw devices for the surfaces must be supported.

- An **fsck**[4] of every mounted disk surface should not be required after a power failure, a potentially lengthy process.

Driver Architecture

The implementations consists of two parts, the surface driver and the changer driver (see the figure 3). The changer driver controls the physical motion of the picker inside the autochanger. Requests to surface devices are routed by the file system switch table to the surface driver. If the surface driver determines that media movement is required, a request is sent to the changer driver. Once the cartridge is in the drive, a disk request is generated by the surface driver and passed on to the normal disk device driver [4, 5] for execution. By separating the disk and changer functions, drives and autochangers with different interfaces can be supported in the future.

The Swapping Algorithm

The driver must orchestrate the swapping of media in and out of the drives in such a way that it looks like each piece of media has its own drive, and with reasonable performance. The implementation needs to make sure that no process is starved, as well as insuring throughput is not compromised (i.e., it must trade off throughput and response time).

Starvation Avoidance

When requests for different surfaces occur, only one of those surfaces can be inserted into each drive. The other requests must be suspended. To avoid these suspended requests waiting forever, some limit must be imposed on the duration a cartridge can be in a drive if other requests are waiting.

Our solution is to set a time limit, called the hog time, which a cartridge can be in a drive processing requests, while other requests are waiting. Once this time expires, that cartridge is removed and the one waiting the longest is inserted. Cartridges continue to be swapped in this round-robin fashion. If requests no longer arrive for the inserted surface, it is removed and a new cartridge inserted. Increasing the value of the hog time reduces the response time for requests on other cartridges while increasing the throughput for the process with its cartridge in a drive. Reducing its value will decrease the response time for the users waiting for a drive but can cause excessive swapping and reduce total throughput.

Obviously, some compromise must be made in the selection of the hog time. The rule-of-thumb we discovered was that this time should be somewhat larger than the time required to exchange a cartridge. Further increasing this depends upon the type of access being performed (e.g., listing directories vs copying large graphics files). In our implementation, a 20 second limit was satisfactory.

Delay between requests

One problem with the round-robin algorithm described above is that when a request for a cartridge currently inserted into a drive is finished, there may not be any additional requests for that surface. If other requests were pending, an new cartridge would be inserted in its place. However, shortly after the cartridge exchange is started, a request for the original surface could arrive. This could result in a cartridge swap for every request.

To avoid this problem an additional time limit was added called the wait time. This is the duration a cartridge can reside idle in a drive while other requests are waiting to use the drive. If no requests

[4] An fsck returns the file system data structures to a consistent state after a power failure.

arrive for an inserted cartridge within the wait time, and other requests are waiting for the drive, the cartridge is removed, and another surface that is pending is then inserted.

Choosing a wait time too large can add extra dead time before swapping the cartridge out, essentially increasing the effective swap time of the autochanger. Making the wait time too small increases the chances that consecutive requests for the same surface could then cause thrashing. We found a wait time of one second to be sufficient to avoid the extra swapping in most instances.

Because certain configurations will require different hog and wait times, these values are configurable (via drive ioctl's) while the system is running. For example, in a backup solution, the hog time should probably be set high so that other processes that make requests to the autochanger won't disrupt the throughput too much. Likewise, if an application program reads blocks of data then processes that data for more than a second before it reads more data, the wait time should be increased to avoid possibly swapping between consecutive reads.

Major Design Problems

There are three problems that the design of the driver must overcome. 1) Avoiding an **fsck** of all the cartridges after a power loss, 2) Avoiding excessive swapping caused by the syncer, and, 3) Support the concept of asynchronous read and write operations.

Problem: Avoiding a Lengthy FSCK

The superuser must execute the **mount** command for each surface to be accessed before users can perform file system operations to the autochanger. Normal winchesters require a file system check (the **fsck** command) if the power cycles while the disk is mounted. Since we do not want to require the user to **fsck** every surface, which could take several hours, but still want it to be available to users, we must make sure the file system on the disk is in a known, valid state every time the media is removed from a drive.

Solution

The autochanger driver uses the concept of *virtually* mounted devices. A virtually mounted device is not vulnerable to power failure corruption of its file system. When a device is virtually mounted, it is not directly connected into the rest of the file system, only enough information about the file system and device are stored to allow it to appear to be mounted. The file system is now *available* for use. However, since it is not directly connected to the rest of the file system, files can not be actually accessed until the device has been physically mounted.

Cartridges in the autochanger which are waiting in their slots will only be virtually mounted. When a file on a virtually mounted device is referenced by a user, the file system code uses the stored information to physically mount the device before the file is accessed. When the operations on that device are finished, the file system will physically unmount the device. This causes any buffers in the host that were not written to the device to be flushed. The file system is now in a state that would not require file system repair on a power failure. The cartridges only need to be virtually mounted when the system is first booted. The cartridges will be physically mounted when they are inserted into a drive and unmounted each time they are removed.

The result of virtually mounting the cartridges is that, 1) all the file systems on the cartridges are available to users, 2) no superuser interaction is required to mount them each time a cartridge is swapped, and 3) any of the cartridges waiting in slots to be accessed do not need to be file system checked if power is lost to the system.

File System Interaction

In order to support virtually mounted devices, a mechanism must be created to allow the file system to tell the autochanger driver what functions to perform when it is time to physically mount and unmount a surface. This was done by adding an extra entry point in the bdevsw[] table. This table is indexed off the major number of the drivers and contains the entry points for all the drivers. The cdevsw[] table is unchanged. The entry point added will be called by the mount_vfs entry point[3] for the file system (i.e., at virtual mount time). The parameters of the routine are the virtual file system pointer and two pointers to functions. The first function (<fs_type>_punmount) is called when a cartridge with that file system type is to be removed from a drive, and the second (called <fstype>_pmount) when it is to be inserted. When the file system is unmounted by the user (i.e., a virtual unmount), the same entry point is called but with null pointers as parameters.

The advantages of this method are as follows:

- Each file system type can tailor the physical mount/unmount code to best match that particular file system's implementation. If the file system does not require any operations to be performed on the media (such as the CD-ROM file system [6]), the file system can ignore the entry point. New file systems can be added without requiring changes in the autochanger driver. Only the pmount and pmount routines need to be provided if desired.

- New autochanger drivers can be added without forcing file system changes. They can just use the same pmount and pumount functions.

- Different types of file systems with different physical mount and unmount requirements can reside in the same autochanger and be mounted at the same time.
- In addition, an unmount of a virtually mounted disk will not require the cartridge to be inserted. This is especially useful when shutting down a system which has many autochanger cartridges mounted.

Problem: The Syncer Causing Extra Swaps

In order to limit the number of dirty buffers which haven't been written to the disk at the time a power failure occurs, there is a process that executes periodically that flushes these buffers. This process, called the syncer, schedules up all the dirty buffers to the drivers so they will be written out. If a surface is removed from the drive without flushing all the dirty buffers, when the syncer executes the cartridge will have to be reinserted. Having several cartridges mounted can cause excessive thrashing every time the syncer executes.

Solution

The solution is to write out all the dirty buffers for the surface before the cartridge is removed. This is already what is being done as part of the physical unmounting process. All the dirty buffers are flushed.

Problem: Supporting Asynchronous Operations

Every block device driver in the unix kernel must be able to support the concept of asynchronous requests. An asynchronous requests essentially means that the file system can make a request to the drive to perform for instance, a write to disk but does not want that driver to sleep, it should queue the request and immediately return without actually waiting for the I/O to complete. This is done through the strategy entry point in the driver. With the autochanger, this means a design criteria is that an asynchronous request to a surface that is not in a drive at the time must not sleep.

This tends to pose a problem in that the request can go ahead and get queued up, however once it is queued some thread of execution must eventually perform that process. But with the way we have architected this as a pseudo driver, the normal method of doing this i.e., through interrupt calls through the normal driver, can not be performed.

In a normal disk driver, the top end performs the queuing of all the requests though the strategy entry point. The bottom end will start a request and use its interrupts to provide the extra thread of execution to perform the asynchronous operations until the queue is empty. While the bottom end is flushing, the top end is queuing them up. But with the autochanger, since we do not have an interrupt structure to provide that extra thread of execution to flush those asynchronous requests, another method must be chosen.

Solution

There are two basic ways to solve this problem. One is provide the extra thread of execution through the disk drivers interrupt calls. This adds extra complexity and coupling to both the drivers. The method we have chosen is to create extra kernel daemons to provide the extra threads of execution necessary. Two daemons are used, a transport daemon which is responsible for moving the cartridges, and a spinup daemon responsible for spinning up the drive. The reason we have two daemons is to overlap moves and spinups. For example, after a cartridge has been put into a drive and begins to spinup, the picker can be used to move another cartridge out of another drive.

Overall Structure

Figure 4 shows the overall structure and the major pieces of the autochanger driver. Each circle represents a separate thread of execution.

ACCEPT REQUESTS
: Executes any requests for surfaces already in drives. All other requests are queued. In charge of determining if a swap is to occur as the result of receiving a request by checking the hog time, etc.

TRANSPORT DAEMON
: Only runs when ACCEPT REQUESTS determines a swap should occur. This also performs the physical unmounting to make sure the surface in the drive is put in a consistent state before it is removed from the drive.

SPINUP DAEMON
: In charge of spinning up the cartridge after it has been inserted. It also performs the physical mounting and processes any asynchronous requests waiting to use that drive.

SCHEDULE ASYNC REQUEST
: This process is executed when no synchronous requests are waiting for a surface. This is necessary due to the requirement of asynchronous requests needing to return without waiting for any I/O to be performed. This is not a daemon but is called under interrupt from the wait_timer started in the spinup daemon.

CONTROL
: The vertical line on the drawing represents a control structure. This is where the wait timer is started. If it goes off, SCHEDULE ASYNC REQUEST is started if no

synchronous requests are waiting to use the drive.

The way these daemons solve the asynchronous write problem is that the transport daemon, in addition to its responsibility of moving the cartridges around also flushes the asynchronous write operations to the disk before it removes a cartridge from a drive. The spinup daemon likewise, is also responsible for flushing all the waiting asynchronous requests to a new cartridge that has just been put into a drive. That way all the asynchronous writes will be flushed out to a drive.

If an asynchronous request arrives during the time a cartridge is currently inserted into a drive, that request will just be passed to the disk driver immediately. Therefore, this maintains the asynchronous function of the autochanger driver. One advantage of using this daemon approach is that it is portable to other architectures of unix.

User Interaction Problems with the Autochanger Itself

Most autochangers offer a method of inserting and ejecting cartridges through some sort of mail slot. The ability of the user to eject cartridges through this mail slot causes new user interaction problems.

Problem: Users Loading and Ejecting Media Which is in Use

A user should not be able to walk over to the autochanger and eject a cartridge that is currently in use (e.g., mounted). Likewise, he should be able to eject a cartridge that no one is currently using. This can result in data loss or could result in a system panic.

Solution

The autochanger we integrated has a SCSI interface [2]. The SCSI autochanger specification defines a concept where each cartridge can be reserved by a host, effectively disallowing the removal of a cartridge from the autochanger. To accomplish this, each cartridge is reserved when it is first

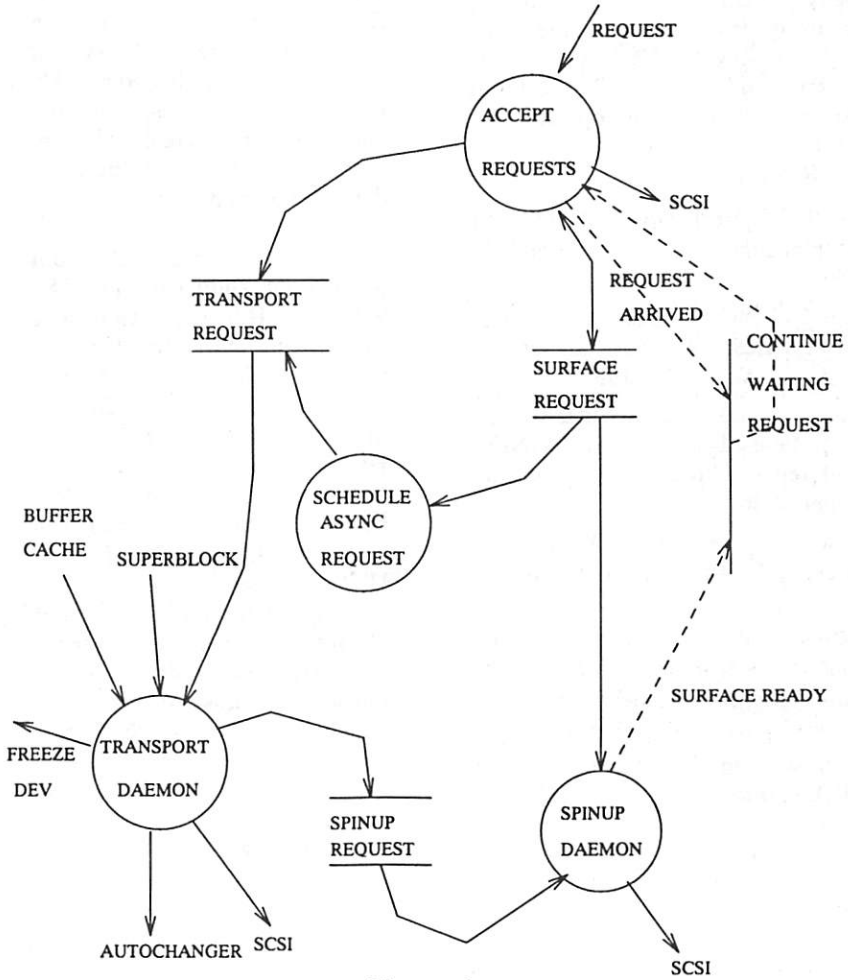


Figure 4

opened and released on the last close. The autochanger will not allow anyone to eject that cartridge while it is in use. Cartridges that are not opened can still be removed via the mailslot.

Another way to secure the autochanger is to send the prevent/allow command to the autochanger itself. This SCSI command effectively prevents anyone from using the mailslot. No cartridges can be removed or inserted into the autochanger.

Conclusion

The HP C1700A autochanger has been successfully integrated onto the HP Series 300 and 800 computer systems. Neither existing commands nor application programs required modification to maintain their functionality. The file systems residing on cartridges maintained their NFS functionality with other machines on the network. The file systems were also protected from power failure to avoid lengthy file system recovery processes.

The autochanger driver has brought the DASS technology to the end user in a simple, straightforward manner. Direct Access Secondary Storage will streamline the process of handling data, eliminating some of the concerns of insufficient storage space and inconvenient file retrieval. Vast amounts of information will be conveniently accessible, offering new freedom and security in computing operations.

References

[1] 130mm Rewritable Optical Disk Cartridge for Information Interchange, Formats A and B. ISO-DIS 10089.

[2] Small Computer System Interface - 2 (SCSI-2), Draft proposed American National Standard for Information Systems, August 22, 1989.

[3] S. R. Kleiman, "Vnodes: An Architecture for Multiple File System Types in Sun UNIX", USENIX Conference Proceedings, Atlanta, Georgia, (Summer 1986)

[4] M. J. Bach, *The Design of the UNIX Operating System*, Prentice-Hall, Englewood Cliffs, NJ (1986).

[5] Loeffler, McKusick, Karels, and Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System*, Addison-Wesley, 1989.

[6] Ping-Hui Kao, "Support of the ISO-9660/HSG CD-ROM File System in HP-UX", *Proceedings of the USENIX Association Summer Conference*, 1989.

Bruce Thompson graduated from Iowa State University in 1981 with a B.S. in Computer Engineering. He joined Hewlett-Packard's Greeley Storage Division after graduation to work on firmware development for 1/2" tape drive products. Other projects include the design of the DDS format standard for digital audio tapes, a disk driver for PC's, and the CD-ROM file system for HP-UX. During his spare time he enjoys amateur radio and portrait and wedding photography. His electronic mail address is brucet@hpgrla.hp.com.

With Hewlett Packard since 1980, Daryl Stolte earned a BSME degree from Cal Poly at San Luis Obispo, California. After discovering that software is the wave of the future, he completed a BSCS degree. Daryl continues to work on autochanger related system software and applications. He lives in Fort Collins, Colorado and enjoys volleyball, windsurfing and stereo photography. His electronic mail address is daryls@hpgrla.hp.com.

David Ellis graduated from Iowa State University in 1981 with a B.S. in Computer Engineering. He has been a member of the technical staff at Hewlett-Packard's Greeley Storage Division for 7 years. His work there has included the firmware design and system integration of their half-inch tape products. He learned UNIX 12 years ago and has been using it ever since. His current responsibility is porting the optical autochanger driver to other operating systems. During his spare time he enjoys singing barbershop quartet music and downhill skiing. His electronic mail address is davee@hpgrla.hp.com.

Reach all authors at Hewlett Packard Greeley Storage Division; 700 71st Avenue; Greeley, Colorado 80634.

USENIX Summer Conference
June 11-15, 1990
Anaheim, California

Douglas Comer, James Griffioen –
Purdue University

A New Design for Distributed Systems: The Remote Memory Model

ABSTRACT

This paper describes a new model for constructing distributed systems called the *Remote Memory Model*. The remote memory model consists of several client machines, one or more dedicated machines called *remote memory servers*, and a communication channel interconnecting them. In the remote memory model, client machines share the memory resources located on the remote memory server. Client machines that exhaust their local memory move portions of their address space to the remote memory server and retrieve pieces as needed. Because the remote memory server uses a machine-independent protocol to communicate with client machines, the remote memory server can support multiple heterogeneous client machines simultaneously.

This paper describes the remote memory model and discusses the advantages and issues of systems that use this model. It examines the design of a highly efficient, reliable, machine-independent protocol used by the remote memory server to communicate with the client machines. It also outlines the algorithms and data structures employed by the remote memory server to efficiently locate the data stored on the server. Finally, it presents measurements of a prototype implementation that clearly demonstrate the viability and competitive performance of the remote memory model.

Background

Virtual memory provides a necessary abstraction for writing architecture-independent portable programs. In a virtual memory architecture, each program has a large, linear address space in which it places code and data. Applications may use more memory than physically available, freeing the programmer from the responsibility of physical memory management. The operating system creates the virtual memory illusion using secondary memory for backing storage when the system exhausts physical memory.

Many conventional virtual memory systems use random access magnetic disks for backing storage[8]. High data transfer rates, random access capabilities, and large capacity make disks a desirable form of backing storage. Many virtual memory systems use demand paging to retrieve data from secondary storage on demand when the system accesses the data. The desire to service page faults quickly makes a disk's fast random access capability extremely valuable. In addition, most operating systems use disks both for backing storage and file storage.

Because disks have become such a prominent form of backing storage, we almost always associate backing storage with disks. Even distributed systems containing diskless machines use remote disks for backing storage. Instead of connecting a disk to each workstation, many distributed systems allow diskless clients to share a disk resource over the network through a special file server machine. Sun Microsystem's SunOS running on a diskless workstation accesses a remote NFS file on disk for backing storage[1,10]. Similarly, the Sprite operating system uses a file on its remote file system for virtual memory backing storage[7,12]. The Mach and Chorus operating systems allow users to define their own backing store and paging methods[3,11], however, most implementations still use a local disk for backing storage.

Equating disks with backing storage impacts the way we design virtual memory operating systems. Operating systems use several techniques to minimize the time spent moving memory to and from backing storage. To optimize head movement on the disk, the operating system often groups write operations together and issues them all at once. To avoid extra bus traffic and disk activity, the operating system delays moving memory to backing storage until absolutely necessary.

Clearly, technology has significantly impacted the way we design systems. This paper presents a new model for designing distributed systems based on remote memory backing storage. The paper describes the impact remote memory has on the design and functionality of such a system and presents a prototype implementation along with experimental results.

The Remote Memory Model

The *remote memory model* provides a new basis for designing distributed systems. The model consists of several client machines, various server machines, one or more dedicated machines called *remote memory servers*, and a communication channel interconnecting all the machines. Figure 1 illustrates one possible remote memory model architecture. In the remote memory model, client machines use remote memory for backing storage rather than disks. The remote memory server provides a shared resource for all client machines. Clients use the communication channel to access memory on the remote memory server.

Figure 1 pictures several diskless client machines connected to a local area network (LAN). The diskless client machines access a remote file server for file storage. Client machines may use a local disk or a remote file server for file storage, but, in either case, the client machines use the remote memory server for backing storage. Each client's operating system supports multiple users in separate virtual address spaces. The operating system also provides the functionality needed to access remote memory.

Each client machine has a private local memory large enough to support the usual processing demands of the client. The remote memory server provides additional memory for applications requiring exceptionally large amounts of memory. When the client exhausts the local physical memory, the operating system moves data to the remote memory server, and retrieves the data as needed.

To keep the model as general as possible, we assume the system consists of heterogeneous machines. Each remote memory server provides remote memory backing storage for multiple heterogeneous client machines simultaneously. The system may contain several remote memory servers. Although the model, in its most general form, does not prohibit a client from communicating with more than one remote memory server, we simplify the model by assuming that each client machine only accesses one predefined remote memory server. Each remote memory server provides a large physical memory resource used to store client data. Remote memory servers may use a local disk to provide additional storage capacity, if needed.

The communication channel allows client machines to send and receive data to and from the remote memory server and other servers. We assume the communication channel has a relatively high bandwidth and low delay (for example, an 10Mb/s Ethernet). To encompass as many communication technologies as possible, we assume the communication channel provides unreliable packet delivery.

The remote memory server makes the remote memory model uniquely different from conventional distributed systems. In conventional distributed systems, each client's virtual memory system is completely independent from all of the other client's virtual memory systems. Even though clients access a common disk via a file server, their virtual memory systems do not interact in any way. The remote memory model takes a different approach, viewing the memory on the remote memory server as a part of the total memory available to the system. In the remote memory model, the remote memory server dynamically allocates regions of its memory to clients that require additional memory space, continually reassigning portions of its memory to clients according to their needs. Clients share the memory space available on the server with all other client machines. As a result, the remote memory server expands the memory available to each client machine. This difference from conventional system gives the remote memory model several desirable

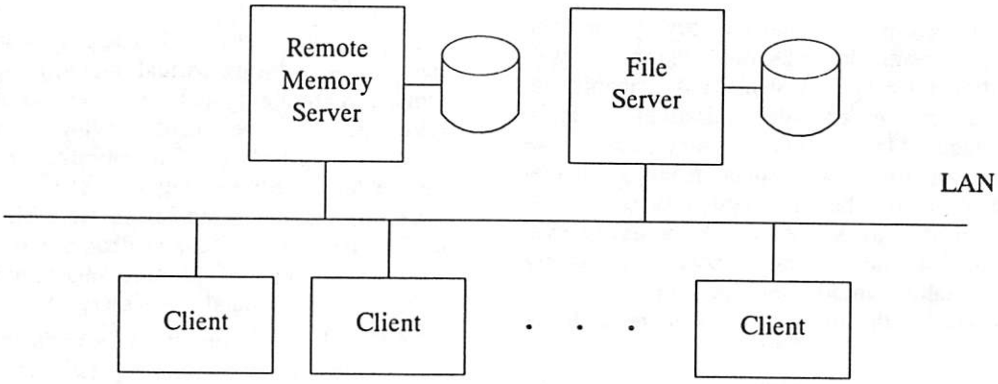


Figure 1: An Example Remote Memory Model Architecture

properties.

Additional Memory: Because clients share a large remote memory resource, client machines may obtain additional memory for applications that require large amounts of memory. We assume clients have enough local memory to support the usual processing demands, but request additional memory space from the remote memory server for large applications.

Arbitrarily Large Storage Capacity: The memory server may employ a form of virtual memory to present an arbitrarily large memory resource to the client machines. The server may connect to one or more disk drives and use a memory replacement scheme to substantially enlarge the storage capacity of the server. As a result, the server presents a large memory space to clients even though the server has a finite physical memory capacity. When the clients exhaust the physical memory on the server, the server moves some of the client's data to disk, making room for more client data in physical memory. Client machines do not know the size of the physical memory on the remote memory server and do not need to restrict application memory usage in any way. From the client's viewpoint, the server simply provides a large memory resource.

Data Sharing: The remote memory server provides a centralized memory, allowing homogeneous client machines to efficiently share data. Although the remote memory model permits data sharing between clients via the remote memory server, the model does not specify the mechanism used to share data. Depending on the type of data sharing desired, the remote memory server may implement mechanisms that allow read only sharing, read-write sharing, or no sharing[6]. Because homogeneous client machines often execute the same applications, use the same shared libraries, and execute the same kernel code, even simple data sharing mechanisms can significantly reduce the amount of server memory used by client machines.

Offloading File Server: The remote memory model improves file system performance by removing paging activity from the file system. Removing paging activity from the file system significantly reduces contention for the disk and eliminates many extra head movement operations. Paging activity tends to access data regions in a random fashion, while file activity tends to access data sequentially. Because the nature of paging activity differs from the nature of file activity, separating paging activity from file activity allows us to implement each operation efficiently. Unlike a remote file server, the remote memory server understands paging activity and makes intelligent decisions regarding storage and retrieval of the data.

Remote Memory Semantics: The remote memory model allows us to define the remote memory semantics in a way that meets the reliability requirements of the system as a whole. For example, we may view the memory on the remote memory server as one component of the total system memory. If any part of the system's memory fails to operate correctly, such as the client's local memory or the remote memory server's memory, the system may lose valuable data. Given these semantics, we do not require the remote memory server to maintain multiple or permanent copies of the data. A different definition of the remote memory semantics may require the server to provide reliable storage. In this case, the remote memory server must maintain duplicate copies of all data, either on other servers or on a permanent storage device such as a disk. Still another definition may require remote memory to provide reliable storage and reliable retrieval. In this case, multiple servers may cooperate to insure that clients can access remote memory at all times.

The Design

The prototype design concentrates on three components of the system: the client virtual memory operating system, the remote memory server, and the protocol clients and servers use to communicate. The design goals we wanted to achieve were:

- to design a communication protocol independent of the underlying network architecture.
- to design a communication protocol that guarantees reliable delivery.
- to make the remote memory resource available to heterogeneous client architectures and heterogeneous client operating systems simultaneously.
- to create a system that provides efficient interaction between client and server without sacrificing the generality mentioned in the previous design goals.

As a result, we designed a protocol that provides reliability, architecture independence, and efficiency, along with a remote memory server capable of efficiently supporting multiple heterogeneous client architectures simultaneously.

The Remote Memory Communication Protocol

The remote memory communication protocol used between clients and a remote memory server consists of two layers: the Xinu Paging Protocol (XPP) layer, and the Negative Acknowledgement Fragmentation Protocol (NAFP) layer.

The XPP Layer

Client operating systems use XPP to reliably transfer memory to and from the remote memory server. XPP supports four basic message types: page store requests, page fetch requests, process create requests, and process terminate requests. When a client machine exceeds the capacity of the local physical memory, the client issues a page store request to store data on the remote memory server. Later, the client issues a page fetch request to retrieve the data from the server. When a client creates or terminates a process, it informs the remote memory server using an create request or a terminate request which we describe in greater detail in section 3.2.3. All XPP request messages originate at the client. The server accepts XPP messages, processes them, and sends reply messages.

XPP provides a communication mechanism independent of the client architectures comprising the remote memory model, permitting heterogeneous client machines to communicate with a single remote memory server. Because the virtual memory system on each client has its own page size, XPP page store and page fetch requests allow clients to transfer variable size memory regions to and from the server. XPP transfers pages of any size, regardless of the underlying communication channel's transport characteristics or maximum packet size.

The paging activity between a client and a server requires reliable, in-order, deliver of all messages. If XPP did not reliably deliver messages in-order, a store request followed by a fetch request could arrive out of order, or a store request could be lost, causing incorrect results. XPP employs sequence numbers, positive acknowledgement, timeouts, and retransmissions to insure reliable, in-order, processing of XPP messages.

Each client machine assigns a sequence number to every XPP message it sends. The remote memory server remembers the sequence number of the last XPP message received from each client machine. The sequence number serves two purposes: it uniquely identifies each message and imposes an ordering on the list of messages. Although the sequence numbers define a processing order that insures correctness, the system does not need to impose such a strict ordering on message processing to achieve correct results. For example, the server may process a message for page i before or after a message for page j and still obtain correct results. However, if a client sends a store request for page i followed by a fetch request for page i, the server must process the store request before the fetch request. XPP achieves correctness by defining a partial ordering on the list of messages. In addition to a sequence number, each XPP message contains a *preceeding message number*. The preceeding message number specifies the sequence number of the most recent preceeding message that must be processed before the current message can be processed. The server may process any message as long as it has already processed the associated preceeding message.

XPP uses end-to-end positive acknowledgements (ACKs) to indicate that the server performed the requested operation[9]. XPP does not use acknowledgements to indicate that a message was successfully transmitted like many other protocols do. Instead, an XPP ACK provides an end-to-end acknowledgement, signaling the successful completion of the requested high level operation. When the server completes the requested operation, it sends a positive acknowledgement to the client containing the results of the operation. Client machines do not discard local copies of pages until the server acknowledges that it has stored the data in remote memory. Most XPP ACKs simply indicate success or failure; however, page fetch replies contain the requested data in addition to the status field.

XPP guarantees reliable delivery using timeouts and retransmissions. The client machine timestamps each message sent to the server, and then holds the message until the server replies with an XPP ACK. If the client does not receive an ACK within a predefined timeout period, the client resends the message. Because all messages originate on the client side, the server never initiates a request and does not need to implement the timeout/retransmission mechanism, greatly simplifying the implementation and improving efficiency.

The NAFP Layer

The Negative Acknowledgement Fragmentation Protocol (NAFP) provides the support needed to efficiently transport XPP messages over a wide variety of communication channels. To allow operation over as many network architectures as possible, the remote memory communication protocol assumes, as a minimal requirement, that the communication channel provides unreliable datagram service.

Because XPP supports a wide variety of page sizes, the length of an XPP message may exceed the maximum packet size of the underlying physical communication channel. NAFP accepts an entire XPP message and breaks the message into fragments. NAFP then transmits each fragment over the communication channel, reassembles the fragments into a complete message, and presents the message to the receiving XPP layer. Because NAFP and XPP adhere to the network layering principle, the sending and receiving XPP layers see exactly the same message[4].

Although conventional communication channels, for instance local area networks, provide reasonably reliable packet delivery, they still drop packets, deliver packets out of order, deliver packets late, or corrupt packet contents. The XPP layer corrects such errors and guarantees reliable delivery. However, the XPP reliability mechanism usually detects communication errors long after the error occurs. XPP then pays a high cost to correct the error. For example, to send or receive an 8K byte page from a Sun3/50 over an Ethernet with a maximum transmission unit (MTU) of approximately 1500 bytes requires a minimum of 6 packets. If the communication channel loses or corrupts any 1 of the 6 packets, XPP will not detect the error until the timeout occurs, and then it must resend all 6 fragments. Because XPP guarantees reliable delivery, NAFP does not need to correct any errors. However, to improve efficiency, NAFP attempts to detect and correct errors as soon as they happen, improving, but not guaranteeing, reliability.

NAFP improves reliability using negative acknowledgements (NACKs). NAFP assigns a sequence number to each fragment and sends each fragment in order. As packets arrive on the receiving end, the NAFP layer reassembles the XPP message but does not acknowledge any of the fragments. As long as no communication errors occur, NAFP transfers messages efficiently with no additional overhead. A fragmentation error arises when a fragment arrives out of order. The receiving end remembers the sequence number of the last fragment for each partially transmitted message. As soon as the NAFP layer receives a fragment out of order, the receiving NAFP layer sends a negative acknowledgement to the sending NAFP layer containing the missing fragment's sequence number. The sending NAFP layer receives the NACK and resends the missing fragment. NAFP does not guarantee reliable delivery of fragments. Instead, NAFP makes a half-hearted attempt to correct errors, sending a single NACK for each missing fragment in hope that the sender will receive the NACK and resend the missing fragment. If the simple, low cost, NAFP error correction mechanism fails, XPP will detect the error and take the corrective measures needed to reliably deliver the message. Because NAFP improves reliability, XPP rarely retransmits messages. In the expected case, in which no communication errors occur, NAFP incurs no additional overhead.

Because the remote memory communication protocol imposes minimal requirements on the communication channel, we can use the remote memory communication protocol over any transport mechanism that provides unreliable datagram service. Almost any protocol can function as the underlying communication protocol, including reliable datagram protocols, stream protocols, or virtual circuit protocols. To achieve independence from the underlying physical network architectures, we use an architecture-independent protocol like UDP or VMTP as the underlying datagram protocol. Architecture independent protocols like UDP and VMTP allow the remote memory communication protocol to operate over local area networks comprised of several different physical network architectures.

The Remote Memory Server

Client performance depends, to a large extent, on the delay the network and the remote memory server introduce. To improve client performance, the remote memory server attempts to minimize the delay by using the remote memory communication protocol and efficient data look-up algorithms. To support as many client machines as possible, the remote memory server avoids preallocation of resources in order to make efficient use of memory.

One of our goals required support for heterogeneous client access to the remote memory resource. Using the remote memory communication protocol, the remote memory server transfers data to and from heterogeneous clients in an architecture-independent manner. The remote memory server supports all architectures regardless of byte size or byte order. Although the remote memory server maintains information about each memory segment it stores, the server does not attempt to interpret or modify the stored data. The server simply returns data in exactly the same form in which it was received.

Efficient Use of Memory

Heterogeneous machines running heterogeneous operating systems use a wide variety of page sizes. The remote memory server uses dynamically allocated data structures to store the variable size memory segments clients send to the server. The remote memory server divides the available memory into small fixed size segments or data blocks. When a client machine sends a page store request to the remote memory server, the server allocates the precise number of data blocks needed to store the page. The tradeoff between data block management overhead and memory space utilization makes it difficult to chose an optimal data block size. Using large data blocks causes internal memory fragmentation, and using small data blocks increases the data block management overhead. In theory, the server defines the data block size as the smallest common denominator of all the client page sizes such that the overhead is still reasonable. In practice, only a few popular page sizes exist based on powers of two, making the choice easy.

The remote memory server allocates data blocks dynamically for each new store request. The data structures do not require the server to store the data in contiguous data blocks. If the server cannot find a set of contiguous data blocks large enough to store the data, the server may spread the data across several disjoint data blocks. The ability to scatter the data from a store request across memory results in efficient use of memory and provides support for a wide variety of page sizes. Because many of the client machines have the same page size, the remote memory server usually allocate and frees segments of the same size, resulting in decreased external memory fragmentation. Consequently, the server can usually allocate contiguous data blocks to each new store request.

Efficient Data Look-Up

To reduce the delay associated with retrieving memory from the remote memory server, the server attempts to minimize the time spent searching the data structures for the desired data. The server uses a *data* hash table and a double hashing algorithm to locate data. Data hash table entries maintain information about client pages stored on the server. Each active hash table entry contains information for exactly one client page, including information regarding the identity of the page, and a range or list of data blocks containing the data. The remote memory server uses a single *data* hash table to store all the data from all the client machines. Client machines uniquely identify a page with an ordered triple consisting of a unique machine identifier, a process identifier, and a page identifier. The server applies a double hashing algorithm to the triple to locate the hash table entry that contains pointers to the data[5]. If the hash table is less than 95% full, the double hashing algorithm, on the average, locates a saved page in less than three probes to the table. As long as the remote memory server limits the utilization of the hash table to less than 95% of the total capacity, the average look-up time remains constant, regardless of the amount of data the clients store on the server or the number of clients using the server.

Memory Reclamation

The remote memory server employs an efficient memory deallocation algorithm to amortize the cost of reclaiming memory over time. The algorithm allows clients to free large amounts of memory with a single inexpensive operation. We assume that most client operating systems support multiple processes and create and terminate processes frequently. To make process termination efficient, client machines require the ability to free large amounts of remote memory in a single operation.

The XPP protocol does not provide a message for releasing individual pages on the server. Instead, XPP provides a terminate process request message. When a process exits, the operating system issues an XPP terminate process request message. The responsibility for freeing all the remote memory associated with the process falls on the remote memory server.

To avoid spending large amounts of time searching for pages associated with the terminated process, the remote memory server maintains a second *process* hash table containing information about all active processes on all client machines. The server maintains a timestamp for each process in the system. When the server receives a page store request for a process, the server saves the process's timestamp with the page in the *data* hash table. Each time the server receives a terminate request, the server updates the timestamp in the process hash table, thereby invalidating all pages associated with the terminated process. The server reclaims obsolete pages during later probes to the data hash table and with a garbage collection process that executes in the background. Each time a probe to the data hash table results in a collision, the server checks the timestamp on the page against the timestamp of the owner. If the timestamps differ, the server reclaims the page. Together, the garbage collection process and the lazy reclamation algorithm amortize the cost of reclaiming memory over time.

A Prototype Implementation and Experimental Results

We designed and implemented a prototype remote memory distributed system based on the remote memory model. The system consists of heterogeneous client machines (Sun Microsystems Sun 3/50's, Digital Equipment Corporation Microvax I's and II's), a remote memory server machine (we have used a Sun 3/50, Vax 11/780, Microvax III, Vaxserver 3100, and an 8 processor Sequent Symmetry as a remote memory server), a file server machine (a Vax 11/780 or Sun 3/50), all connected by a 10 Mb/sec Ethernet. Sun and Microvax client machines simultaneously access the remote memory server for backing storage, demonstrating support for heterogeneous clients.

In the prototype, remote memory is high speed volatile storage, susceptible to failure and data loss. To keep the prototype simple, the remote memory server does not support any data sharing between client machines. After experience with the server, we chose 1K byte blocks as the storage page size.

We built the remote memory communication protocol on top of UDP to allow communication over almost any network architecture. Moreover, UDP allows client machines to reside on a different physical network than the server machine. We have experimented with a configuration in which client machines access a remote memory server on a remote network through several gateways. Even when traversing several gateways to access the remote memory server, the high-cost XPP guaranteed reliability mechanism rarely retransmits messages because the negative acknowledgement fragmentation protocol corrects most communication errors. Another configuration we have used chains remote memory servers together. Client machines access a diskless remote memory server that in turn accesses a remote memory server with a disk.

Our initial timing results show that storing or retrieving an 8K byte segment between a Sun 3/50 client and a Sun 3/50 remote memory server requires an average of 39ms. In contrast, current production systems consisting of diskless Sun 3/50s paging over NFS require an average of 50ms to process an 8K byte read request when accessing a file sequentially. To randomly access an NFS file, as paging activity does, requires an average of 84ms to process an 8K byte read request and an average of 176ms to process an 8K byte write request.

Figure 2 shows the cost of performing a page fetch operation (measured as round trip delay) as a function of the number of requests issued per second by each client machine. We conducted the tests

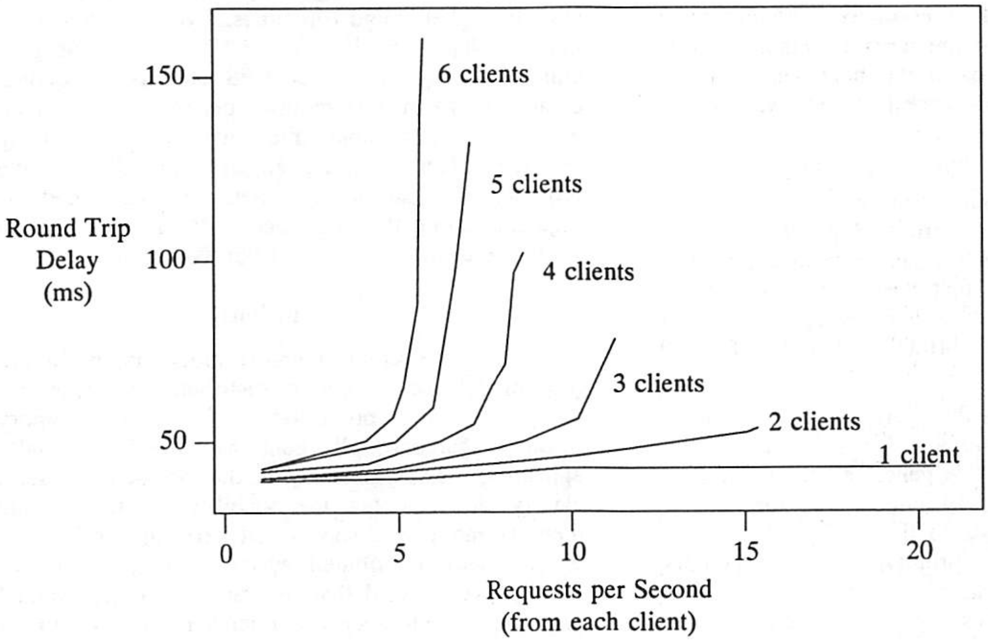


Figure 2: Round Trip Delay As A Function Of The Client Request Rate

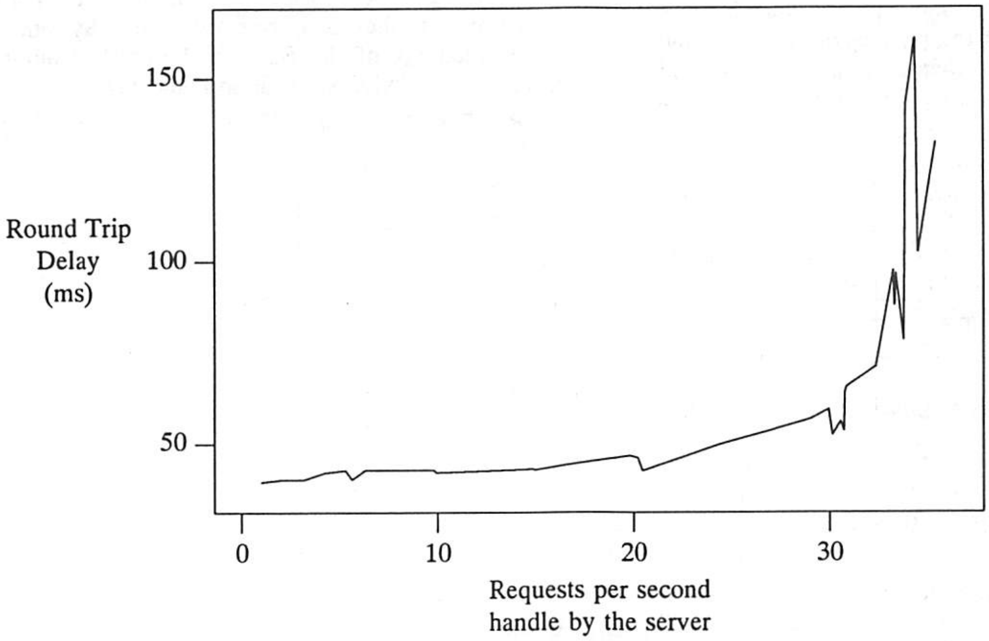


Figure 3: Remote Memory Delay For Various Server Loads

using Sun 3/50 client machines paging to a Sun 3/50 remote memory server and implemented the prototype remote memory server as a UNIX application level process. Because the Ethernet has an MTU of 1500 bytes, the NAFP protocol breaks each 8K byte Sun 3/50 page into 6 Ethernet packets. All client machines send paging requests concurrently. Each client sends requests at a constant rate, uniformly distributed over time, to the remote memory server. The request rates shown in the figure indicate the number of requests per second issued by a single client machine.

The figure shows the round trip delay for a varying number of clients and request rates. The sudden rise in the round trip delay shown in the curves for 3 or more clients can be misleading. The following figure shows that the sudden increase in each curve occurs at the point where the server becomes overloaded (total load of 30 requests/second).

Figure 3 illustrates the average round trip delay as a function of the number of requests the server processes per second. We generate the server load by varying the number of clients and the rate at which they send requests to the server. The figure shows that the current prototype remote memory server, executing as a user level process on a Sun 3/50, efficiently handles up to 30 requests per second. At 30 requests per second the prototype memory server becomes saturated and any more load on the server significantly increases the round trip delay, explaining the sharp rise in the curves pictured in figure 2. For loads of less than 30 requests per second, the average round trip delay remains less than 56ms regardless of the number of clients, If the load on the server is less than 20 requests per second (2/3 of the server's capacity), the round trip times never exceeds 46ms.

If we assume, under usual operating circumstances, that clients send an average of 5 requests per second, then figure 4 illustrates the gradual increase in round trip delay as the number of clients increases to the maximum capacity of the server. As the number of client increases, the number of dropped fragments increases, resulting in slightly higher round trip times. Lyon and Sandberg indicate that 8 diskless Sun 3/50 workstations generate an average load of 30 NFS requests per second, or an average of 3.75 requests per second per client, which includes both file activity and paging activity[2]. Consequently, an average of 5 paging requests per second per client may be somewhat high, indicating that the slope of the line in figure 4 would be even less for the usual workload.

Conclusions

Using the remote memory model as an alternative model for designing distributed systems has many attractive properties. The large memory resource shared by all client machines is especially appealing. Experience with the prototype system clearly demonstrates the viability of the remote memory model and shows that performance is competitive with distributed systems currently in use. Finally, we showed that the remote memory model can support heterogeneous clients machines without sacrificing efficiency.

References

1. R. Sandberg, D. Goldberg, S. Kleiman, Dan Walsh, and Bob Lyon, "Design and Implementation of the Sun Network File System," Proceedings of the Summer USENIX Conference, USENIX Association (June 1985).
2. R. Sandberg and Bob Lyon, "Breaking

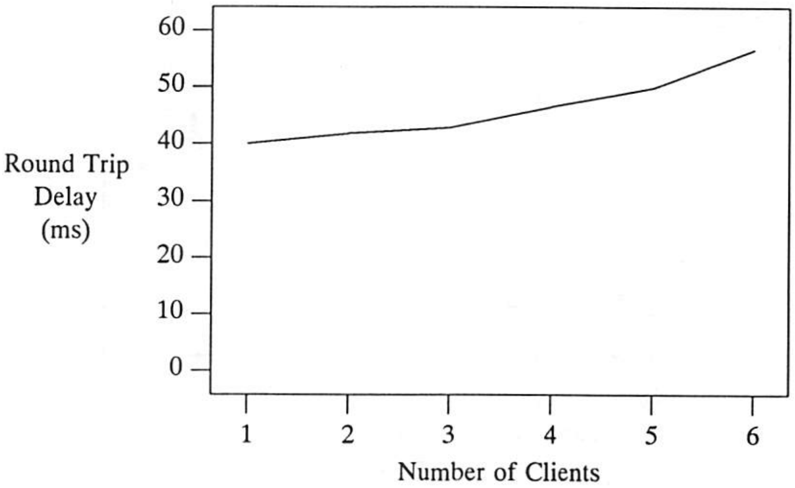


Figure 4: Round Trip Time (Each Client Sends 5 Requests/Second)

Through the NFS Performance Barrier,'' Sun-Tech Journal (August 1989).

3. Vadim Abrossimov and Marc Rozier, ''Generic Virtual Memory Management for Operating System Kernels,'' *Proceeding of the 12th ACM Symposium on Operating System Principles* **23**(5) pp. 123-136 Chorus Systems, (December 1989).
4. Douglas Comer, *Internetworking with TCP/IP: Principles, Protocols, and Architecture,* Prentice Hall (1988).
5. Donald E. Knuth, *Sorting and Searching,* Addison Wesley Publishing Company (1973).
6. Kai Li and Paul Hudak, ''Memory Coherence in Shared Virtual Memory Systems,'' Proceedings of the 5th ACM Symposium of Principles of Distributed Computing (August 1986).
7. John Ousterhout, Andrew Cherenson, Fred Douglis, Michael Nelson, and Brent Welch, ''The Sprite Network Operating System,'' Tech Report UCB/CSD 87/359n, University of California Berkeley (June 1987).
8. James L. Peterson and Abraham Silberschatz, *Operating System Concepts,* Addison Wesley (1985).
9. J. H. Saltzer, D. P. Reed, and D. D. Clark, ''End-To-End Arguments in System Design,'' *ACM Transactions on Computer Systems* **2** pp. 277-288 (1984).
10. Robert A. Gingell and Joseph P. Moran and William A. Shannon, *Virtual Memory Architecture in SunOS,* Sun Microsystems, Inc. (1988).
11. Avadis Tevanian, ''Architecture Independent Virtual Memory Management for Parallel and Distributed Environments: The Mach Approach,'' Tech Report CMU-CS-88-106n, CMU (December 1987).
12. Brent B. Welch, ''The Sprite Remote Procedure Call System,'' Tech Report UCB/CSD 86/302, University of California Berkeley (June 1986).

Dr. Douglas Comer is a full professor in the Computer Science Department at Purdue University where he teaches graduate-level courses in operating systems, internetworking, and distributed systems. He has written numerous research papers and five textbooks, and has been principle investigator on many research projects. He designed and implemented the X25NET and Cypress networks, as well as the Xinu operating system. He heads the Xinu, Cypress, Shadow Editing, and Multiswitch research projects. He is a member of the Internet Research Steering Group and chairman of the Internet Naming Research Group. He is a former member of the CSNET Executive Committee and the Internet Activities Board. Professor Comer teaches networking seminars for Interop Incorporated. He is a member of the ACM, AAAS, and Sigma Xi.

James Griffioen is a PhD candidate in the Computer Science Department at Purdue University. His research interests include operating systems and distributed systems. He has worked on the Xinu and the Shadow Editing research projects. He received an MS degree in computer science from Purdue University in 1988 and a BS degree in computer science from Calvin College in 1985. He received the USENIX scholarship for the 89-90 academic year and is a member of the ACM.

USENIX Summer Conference
June 11-15, 1990
Anaheim, California

A Pageable Memory Based Filesystem

Marshall Kirk McKusick, Michael J. Karels, Keith Bostic – Computer Systems Research Group, University of California at Berkeley

ABSTRACT

This paper describes the motivations for memory-based filesystems. It compares techniques used to implement them and describes the drawbacks of using dedicated memory to support such filesystems. To avoid the drawbacks of using dedicated memory, it discusses building a simple memory-based filesystem in pageable memory. It details the performance characteristics of this filesystem and concludes with areas for future work.

Introduction

This paper describes the motivation for and implementation of a memory-based filesystem. Memory-based filesystems have existed for a long time; they have generally been marketed as RAM disks or sometimes as software packages that use the machine's general purpose memoryWhite, 1980.

A RAM disk is designed to appear like any other disk peripheral connected to a machine. It is normally interfaced to the processor through the I/O bus and is accessed through a device driver similar or sometimes identical to the device driver used for a normal magnetic disk. The device driver sends requests for blocks of data to the device and the requested data is then DMA'ed to or from the requested block. Instead of storing its data on a rotating magnetic disk, the RAM disk stores its data in a large array of random access memory or bubble memory. Thus, the latency of accessing the RAM disk is nearly zero compared to the 15-50 milliseconds of latency incurred when access rotating magnetic media. RAM disks also have the benefit of being able to transfer data at the maximum DMA rate of the system, while disks are typically limited by the rate that the data passes under the disk head.

Software packages simulating RAM disks operate by allocating a fixed partition of the system memory. The software then provides a device driver interface similar to the one described for hardware RAM disks, except that it uses memory-to-memory copy instead of DMA to move the data between the RAM disk and the system buffers, or it maps the contents of the RAM disk into the system buffers. Because the memory used by the RAM disk is not available for other purposes, software RAM-disk solutions are used primarily for machines with limited addressing capabilities such as PC's that do not have an effective way of using the extra memory anyway.

Most software RAM disks lose their contents when the system is powered down or rebooted. The contents can be saved by using battery backed-up memory, by storing critical filesystem data structures in the filesystem, and by running a consistency check program after each reboot. These conditions increase the hardware cost and potentially slow down the speed of the disk. Thus, RAM-disk filesystems are not typically designed to survive power failures; because of their volatility, their usefulness is limited to transient or easily recreated information such as might be found in **/tmp**. Their primary benefit is that they have higher throughput than disk based filesystemsSmith, 1981. This improved throughput is particularly useful for utilities that make heavy use of temporary files, such as compilers. On fast processors, nearly half of the elapsed time for a compilation is spent waiting for synchronous operations required for file creation and deletion. The use of the memory-based filesystem nearly eliminates this waiting time.

Using dedicated memory to exclusively support a RAM disk is a poor use of resources. The overall throughput of the system can be improved by using the memory where it is getting the highest access rate. These needs may shift between supporting process virtual address spaces and caching frequently used disk blocks. If the memory is dedicated to the filesystem, it is better used in a buffer cache. The buffer cache permits faster access to the data because it requires only a single memory-to-memory copy from the kernel to the user process. The use of memory is used in a RAM-disk configuration may require two memory-to-memory copies, one from the RAM disk to the buffer cache, then another copy from the buffer cache to the user process.

The new work being presented in this paper is building a prototype RAM-disk filesystem in pageable memory instead of dedicated memory. The goal is to provide the speed benefits of a RAM disk without paying the performance penalty inherent in dedicating part of the physical memory on the machine to the RAM disk. By building the filesystem in pageable memory, it competes with other processes for the available memory. When memory runs short, the paging system pushes its least-recently-used pages to backing store. Being pageable also allows the filesystem to be much larger than would be practical if it were limited by the amount of physical memory that could be dedicated to that purpose. We typically operate our **/tmp** with 30 to 60 megabytes of space which is larger than the amount of memory on the machine. This configuration allows small files to be accessed quickly, while still allowing **/tmp** to be used for big files, although at a speed more typical of normal, disk-based filesystems.

An alternative to building a memory-based filesystem would be to have a filesystem that never did operations synchronously and never flushed its dirty buffers to disk. However, we believe that such a filesystem would either use a disproportionately large percentage of the buffer cache space, to the detriment of other filesystems, or would require the paging system to flush its dirty pages. Waiting for other filesystems to push dirty pages subjects them to delays while waiting for the pages to be written. We await the results of others trying this approachOhta Tezuka, 1990.

Implementation

The current implementation took less time to write than did this paper. It consists of 560 lines of kernel code (1.7K text + data) and some minor modifications to the program that builds disk based filesystems, *newfs*. A condensed version of the kernel code for the memory-based filesystem are reproduced in Appendix A.

A filesystem is created by invoking the modified *newfs*, with an option telling it to create a memory-based filesystem. It allocates a section of virtual address space of the requested size and builds a filesystem in the memory instead of on a disk partition. When built, it does a *mount* system call specifying a filesystem type of MFS (Memory File System). The auxiliary data parameter to the mount call specifies a pointer to the base of the memory in which it has built the filesystem. (The auxiliary data parameter used by the local filesystem, *ufs*, specifies the block device containing the filesystem.)

The mount system call allocates and initializes a mount table entry and then calls the filesystem-specific mount routine. The filesystem-specific routine is responsible for doing the mount and initializing the filesystem-specific portion of the mount table entry. The memory-based filesystem-specific mount routine, *mfs_mount*(), is shown in Appendix A. It allocates a block-device vnode to represent the memory disk device. In the private area of this vnode it stores the base address of the filesystem and the process identifier of the *newfs* process for later reference when doing I/O. It also initializes an I/O list that it uses to record outstanding I/O requests. It can then call the *ufs* filesystem mount routine, passing the special block-device vnode that it has created instead of the usual disk block-device vnode. The mount proceeds just as any other local mount, except that requests to read from the block device are vectored through *mfs_strategy*() (described below) instead of the usual *spec_strategy*() block device I/O function. When the mount is completed, *mfs_mount*() does not return as most other filesystem mount functions do; instead it sleeps in the kernel awaiting I/O requests. Each time an I/O request is posted for the filesystem, a wakeup is issued for the corresponding *newfs* process. When awakened, the process checks for requests on its buffer list. A read request is serviced by copying data from the section of the *newfs* address space corresponding to the requested disk block to the kernel buffer. Similarly a write request is serviced by copying data to the section of the *newfs* address space corresponding to the requested disk block from the kernel buffer. When all the requests have been serviced, the *newfs* process returns to sleep to await more requests.

Once mounted, all operations on files in the memory-based filesystem are handled by the *ufs* filesystem code until they get to the point where the filesystem needs to do I/O on the device. Here, the filesystem encounters the second piece of the memory-based filesystem. Instead of calling the special-device strategy routine, it calls the memory-based strategy routine, *mfs_strategy*(). Usually, the request is serviced by linking the buffer onto the I/O list for the memory-based filesystem vnode and sending a wakeup to the *newfs* process. This wakeup results in a context-switch to the *newfs* process, which does a copyin or copyout as described above. The strategy routine must be careful to check whether the I/O request is coming from the *newfs* process itself, however. Such requests happen during mount and unmount operations, when the kernel is reading and writing the superblock. Here, *mfs_strategy*() must do the I/O itself to avoid deadlock.

The final piece of kernel code to support the memory-based filesystem is the close routine. After the filesystem has been successfully unmounted, the device close routine is called. For a memory-based filesystem, the device close routine is *mfs_close*(). This routine flushes any pending I/O requests, then sets the I/O list head to a special value that is recognized by the I/O servicing loop in *mfs_mount*() as an indication that the filesystem is unmounted. The *mfs_mount*() routine exits, in turn causing the *newfs* process to exit, resulting in the filesystem vanishing in a cloud of dirty pages.

The paging of the filesystem does not require any additional code beyond that already in the kernel to support virtual memory. The *newfs* process competes with other processes on an equal basis for the machine's available memory. Data pages of the filesystem that have not yet been used are zero-fill-on-demand pages that do not occupy memory, although they currently allocate space in backing store. As long as memory is plentiful, the entire contents of the filesystem remain memory resident. When memory runs short, the oldest pages of *newfs* will be pushed to backing store as part of the normal paging activity. The pages that are pushed usually hold the contents of files that have been created in the memory-based filesystem but have not been recently accessed (or have been deleted)Leffler , 1989.

Performance

The performance of the current memory-based filesystem is determined by the memory-to-memory copy speed of the processor. Empirically we find that the throughput is about 45% of this memory-to-memory copy speed. The basic set of steps for each block written is:

1) memory-to-memory copy from the user process doing the write to a kernel buffer
2) context-switch to the *newfs* process
3) memory-to-memory copy from the kernel buffer to the *newfs* address space
4) context switch back to the writing process

Thus each write requires at least two memory-to-memory copies accounting for about 90% of the CPU time. The remaining 10% is consumed in the context switches and the filesystem allocation and block location code. The actual context switch count is really only about half of the worst case outlined above because read-ahead and write-behind allow multiple blocks to be handled with each context switch.

On the six-MIPS CCI Power 6/32 machine, the raw reading and writing speed is only about twice that of a regular disk-based filesystem. However, for processes that create and delete many files, the speedup is considerably greater. The reason for the speedup is that the filesystem must do two synchronous operations to create a file, first writing the allocated inode to disk, then creating the directory entry. Deleting a file similarly requires at least two synchronous operations. Here, the low latency of the memory-based filesystem is noticeable compared to the disk-based filesystem, as a synchronous operation can be done with just two context switches instead of incurring the disk latency.

Future Work

The most obvious shortcoming of the current implementation is that filesystem blocks are copied twice, once between the *newfs* process' address space and the kernel buffer cache, and once between the kernel buffer and the requesting process. These copies are done in different process contexts, necessitating two context switches per group of I/O requests. These problems arise because of the current inability of the kernel to do page-in operations for an address space other than that of the currently-running process, and the current inconvenience of mapping process-owned pages into the kernel buffer cache. Both of these problems are expected to be solved in the next version of the virtual memory system, and thus we chose not to address them in the current implementation. With the new version of the virtual memory system, we expect to use any part of physical memory as part of the buffer cache, even though it will not be entirely addressable at once within the kernel. In that system, the implementation of a memory-based filesystem that avoids the double copy and context switches will be much easier.

Ideally part of the kernel's address space would reside in pageable memory. Once such a facility is available it would be most efficient to build a memory-based filesystem within the kernel. One potential problem with such a scheme is that many kernels are limited to a small address space (usually a few megabytes). This restriction limits the size of memory-based filesystem that such a machine can support. On such a machine, the kernel can describe a memory-based filesystem that is larger than its address space and use a ''window'' to map the larger filesystem address space into its limited address space. The window would maintain a cache of recently accessed pages. The problem with this scheme is that if the working set of active pages is greater than the size of the window, then much time is spent remapping pages and invalidating translation buffers. Alternatively, a separate address space could be constructed for each memory-based filesystem as in the current implementation, and the memory-resident pages of that address space could be mapped exactly as other cached pages are accessed.

The current system uses the existing local filesystem structures and code to implement the memory-based filesystem. The major advantages of this approach are the sharing of code and the simplicity of the approach. There are several disadvantages, however. One is that the size of the filesystem is fixed at mount time. This means that a fixed number of inodes (files) and data blocks can be supported. Currently, this approach requires enough swap space for the entire filesystem, and prevents expansion and contraction of the filesystem on demand. The current design also prevents the filesystem from taking advantage of the memory-resident character of the filesystem. It would be interesting to explore other filesystem implementations that would be less expensive to execute and that would make better use of the space. For example, the current filesystem structure is optimized for magnetic disks. It includes replicated control structures, "cylinder groups" with separate allocation maps and control structures, and data structures that optimize rotational layout of files. None of this is useful in a memory-based filesystem (at least when the backing store for the filesystem is dynamically allocated and not contiguous on a single disk type). On the other hand, directories could be implemented using dynamically-allocated memory organized as linked lists or trees rather than as files stored in "disk" blocks. Allocation and location of pages for file data might use virtual memory primitives and data structures rather than direct and indirect blocks. A reimplementation along these lines will be considered when the virtual memory system in the current system has been replaced.

References

Leffler , 1989. S. J. Leffler, M. K. McKusick, M. J. Karels, J. S. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System,* Addison-Wesley, Reading, MA (1989).

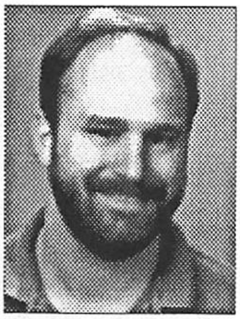
Ohta Tezuka, 1990. Masataka Ohta Hiroshi Tezuka, "A Fast /tmp File System by Async Mount Option," *USENIX Association Conference Proceedings,* pp. 145–150 (June 1990).

Smith, 1981. A. J. Smith, "Bibliography on file and I/O system optimizations and related topics," *Operating Systems Review* **14**(4) pp. 39–54 (October 1981).

White, 1980. R. M. White, "Disk Storage Technology," *Scientific American* **243**(2) pp. 138–148 (August 1980).

Marshall Kirk McKusick got his undergraduate degree in Electrical Engineering from Cornell University. His graduate work was done at the University of California, where he received Masters degrees in Computer Science and Business Administration, and a Ph.D. in the area of programming languages. While at Berkeley he implemented the 4.2BSD fast file system and was involved in implementing the Berkeley Pascal system. He currently is the Research Computer Scientist at the Berkeley Computer Systems Research Group, continuing the development of future versions of Berkeley UNIX. He is president of the USENIX Association, a member of the editorial board of UNIX Review Magazine, and a member of ACM and IEEE.

Michael J. Karels is the Principal Programmer of the Computer Systems Research Group at the University of California, Berkeley. Since the release of 4.2BSD, he has been the system architect for Berkeley UNIX, continuing the development of new versions of BSD. He has worked on all parts of the Berkeley kernel, but has concentrated on the network and TCP/IP implementations. Mike received his B.S. in Microbiology at the University of Notre Dame.



Keith Bostic has been a member of the Berkeley Computer Systems Research Group since 1986. In this capacity, he was the primary architect of the 2.10BSD release of the Berkeley Software Distribution for PDP-11's. He currently has overall responsibility for support of all BSD releases, as well as participating in the continuing development of future versions of Berkeley UNIX. He received his undergraduate degree in Statistics and his Masters degree in Electrical Engineering from The George Washington University.

Appendix A - Implementation Details

```
/*
 * This structure defines the control data for the memory
 * based file system.
 * /
struct mfsnode {
           struct    vnode *mfs_vnode;          /* vnode associated with this mfsnode * /
           caddr_t   mfs_baseoff;               /* base of file system in memory * /
           long      mfs_size;                  /* size of memory file system * /
           pid_t     mfs_pid;                   /* supporting process pid * /
           struct    buf *mfs_buflist;          /* list of I /O requests * /
};

/*
 * Convert between mfsnode pointers and vnode pointers
 * /
#define    VTOMFS(vp)       ((struct mfsnode *)(vp)->v_data)
#define    MFSTOV(mfsp)     ((mfsp)->mfs_vnode)
#define    MFS_EXIT         (struct buf *)-1

/*
 * Arguments to mount MFS
 * /
struct mfs_args {
           char      *name;             /* name to export for statfs * /
           caddr_t   base;              /* base address of file system in memory * /
           u_long    size;              /* size of file system * /
};
```

mfs_mount

```
/*
 * Mount an MFS filesystem.
 */
mfs_mount(mp, path, data)
        struct mount *mp;
        char *path;
        caddr_t data;
{
        struct vnode *devvp;
        struct mfsnode *mfsp;
        struct buf *bp;
        struct mfs_args args;

        /*
         * Create a block device to represent the disk.
         */
        devvp = getnewvnode(VT_MFS, VBLK, &mfs_vnodeops);
        mfsp = VTOMFS(devvp);
        /*
         * Save base address of the filesystem from the supporting process.
         */
        copyin(data, &args, (sizeof mfs_args));
        mfsp->mfs_baseoff = args.base;
        mfsp->mfs_size = args.size;
        /*
         * Record the process identifier of the supporting process.
         */
        mfsp->mfs_pid = u.u_procp->p_pid;
        /*
         * Mount the filesystem.
         */
        mfsp->mfs_buflist = NULL;
        mountfs(devvp, mp);
        /*
         * Loop processing I/O requests.
         */
        while (mfsp->mfs_buflist != MFS_EXIT) {
                while (mfsp->mfs_buflist != NULL) {
                        bp = mfsp->mfs_buflist;
                        mfsp->mfs_buflist = bp->av_forw;
                        offset = mfsp->mfs_baseoff + (bp->b_blkno * DEV_BSIZE);
                        if (bp->b_flags & B_READ)
                                copyin(offset, bp->b_un.b_addr, bp->b_bcount);
                        else /* write_request */
                                copyout(bp->b_un.b_addr, offset, bp->b_bcount);
                        biodone(bp);
                }
                sleep((caddr_t)devvp, PWAIT);
        }
}
```

```
/*
 * If the MFS process requests the I/O then we must do it directly.
 * Otherwise put the request on the list and request the MFS process
 * to be run.
 */
mfs_strategy(bp)                              mfs_strategy
        struct buf *bp;
{
        struct vnode *devvp;
        struct mfsnode *mfsp;
        off_t offset;

        devvp = bp->b_vp;
        mfsp = VTOMFS(devvp);
        if (mfsp->mfs_pid == u.u_procp->p_pid) {
                offset = mfsp->mfs_baseoff + (bp->b_blkno * DEV_BSIZE);
                if (bp->b_flags & B_READ)
                        copyin(offset, bp->b_un.b_addr, bp->b_bcount);
                else /* write_request */
                        copyout(bp->b_un.b_addr, offset, bp->b_bcount);
                biodone(bp);
        } else {
                bp->av_forw = mfsp->mfs_buflist;
                mfsp->mfs_buflist = bp;
                wakeup((caddr_t)bp->b_vp);
        }
}

/*
 * The close routine is called by unmount after the filesystem
 * has been successfully unmounted.
 */
mfs_close(devvp)                                 mfs_close
        struct vnode *devvp;
{
        struct mfsnode *mfsp = VTOMFS(vp);
        struct buf *bp;

        /*
         * Finish any pending I/O requests.
         */
        while (bp = mfsp->mfs_buflist) {
                mfsp->mfs_buflist = bp->av_forw;
                mfs_doio(bp, mfsp->mfs_baseoff);
                wakeup((caddr_t)bp);
        }
        /*
         * Send a request to the filesystem server to exit.
         */
        mfsp->mfs_buflist = MFS_EXIT;
        wakeup((caddr_t)vp);
}
```

USENIX Summer Conference
June 11-15, 1990
Anaheim, California

A Fast /tmp File System by Delay Mount Option

Masataka Ohta – Computer Center, Tokyo Institute of Technology
Hiroshi Tezuka – Sony Computer Science Laboratory Inc.

ABSTRACT

To improve the speed of a /tmp file system, a new mount option **delay** is introduced. A usual UNIX kernel has three types of write operations: sync write, async write and delayed write. For important data such as directory structure, slow but sure sync write is used to maintain file system integrity even through system crash. For less important data, fast delayed write is used to gain efficiency. As delayed write only writes to memory called a buffer cache, which is later copied to the disk, it is much faster than other types of write operations. The **delay** option enforces all write operations on the file system performed with the delayed write. For the /tmp file system, where files are removed shortly after creation, integrity is not so much necessary. So, by specifying the delay option, a fast /tmp file system is obtained. The delay option is implemented with modification of less than 50 lines in UNIX kernel and less than 10 lines in /etc/mount command.

Compared to memory disk implementation of a fast /tmp file system, delay approach is faster. Moreover, with the delay option, /tmp file system may be retained after power failure. The size of the file system is not limited by the size of memory and can be as large as an ordinary file system.

Combined with dynamic buffer caching, delay option can utilize all free memory to accelerate file operations on /tmp.

The delay option is also useful for the fast installation of a totally new file system.

Introduction

Traditionally in UNIX, /tmp (along with /usr/tmp) has been a special purpose directory for temporary files. Temporary files of various compilers, editors and mailers are all created under /tmp.

Files under /tmp are usually created by some command and written some data. But, in such a case, the data is used only internally to the command and the files are removed before the command exits. For commands which performs large amount of IO operations on temporary files, efficient implementation of a file system under /tmp will improve the overall performance.

Or, on several sites, users are encouraged to use /tmp to store large but intermediate data. Such data is often too large for the file systems of users' home directories. So, /tmp is provided as a common work space. For example, in Tokyo Institute of Technology, ETA10 super computer is operated with a 500MB /tmp file system.

Anyway, files under /tmp contains just temporary data and losing them is not a serious problem (except for an editor's work file, which is discussed in 2.2). Actually, on UNIX systems, plain files (or even directories in some case) under /tmp are automatically cleaned up upon system start up. That is, files under /tmp may be lost after a system crash.

Using this special property, there is a room to implement a fast /tmp file system.

Existing approach for the fast /tmp file system is to use a memory disk [1]. A memory disk is a pseudo IO device, which assigns part of main memory as IO storage. As IO requests are executed memory to memory copy, there are no rotational delay nor seek delay. So, fast IO operations are possible.

Of course, the memory disk is volatile, that is, its content is lost after power failure. As stated before, this is not a serious problem for the /tmp file system. But, there is another problem with the memory disk. As memory is much precious resource than disk space, it is not economical to allocate a large amount of memory as a memory disk. Most part of memory should be used for swapping. Moreover, it is practically impossible to allocate a truly large (say 500MB) file system, because the total amount of memory is not so large. So, the memory disk approach is not an ideal solution for a fast /tmp file system.

Delayed Write and Delay Option

UNIX Disk Write Strategy

UNIX kernel accesses file systems through the buffer cache mechanism. A buffer cache is a part of memory which holds a copy of data in disks. As the buffer cache is a cache, it improves IO performance, but the coherence between the buffer cache and the disks must be carefully maintained. So, UNIX kernel has three types of file system write operations: sync write, async write and delayed write [23]. Thus, file system integrity is well assured even after system crash.

As shown in Figure 1a), delayed write is issued according to a write system call and used to write plain data. Data is copied from user process space to a buffer cache, but, no actual IO occurs. Instead, the buffer cache is marked as dirty. Dirty buffer caches are later output to the disk asynchronously by a sync system call. The sync system call is issued once in a 30 seconds, from a system daemon process: /etc/update.

Async write is issued when it is necessary to actually transfer plain data block from buffer caches to disks. That is, when a sync system call is issued, or when there is no free buffer cache. Async write is also issued, when a full block of newly written data is available, because it is very likely that the block is needs actual transfer. As shown in Figure 1b), async write initiates data transfer from the buffer cache to the disk, but dose not wait for the completion. So, async write itself is fast, but async write may initiate unnecessary data transfer if a newly written full block is modified again or removed.

Sync write is issued when the structure of a file system is modified. For example, for the creation and deletion of a file, sync write is used. As shown in Figure 1c), sync write initiates data transfer and wait for the completion. So, even if the system crashes during file system structure modification, loss of integrity is kept minimal. Waiting in sync write is long because it includes disk head seek delay and rotational delay.

Delay Option and Delayed File System

As discussed in the introduction, files under /tmp may be lost after system crash. So, time consuming sync write is not necessary for /tmp. Moreover, async write of a full block is also often unnecessary on /tmp, because it will be removed soon. Instead, issuing delayed write is just enough. So, we introduce a new mount option **delay**. For a file system mounted with **delay** option, no sync write is issued and delayed write is used instead. We call such a file system a delayed file system.

A delayed file system has several obvious advantages over a memory disk.

First, as mentioned in the introduction, the memory disk file system is limited by its size. On the other hand, as a delayed file system is just an ordinary disk file system, it can be as large as desired.

Secondly, delayed file system may survive system crash. As delayed write is finally reflected to disks by a sync system call once in a 30 seconds, if the file system is not modified for 30 seconds, its integrity on a disk is assured. Even if some modification is made within 30 seconds before system crash, something is written on the disk, which will not lost even after power failure, so, automatic recovery by fsck may succeed. Such a property is desirable because ex/vi editor has an automatic mechanism (ex3.7preserve) upon system start up to attempt to salvage workfiles just before system crash.

Implementation effort and actual performance is discussed in the next section.

Evaluation

Implementation Effort

To implement the delay option, UNIX kernel must be modified. Basically, all calls to sync write and unnecessary async write are replaced by conditional statements on a delay option flag which may call delayed write instead. As there are not so many sync/async write calls in the kernel, the delay option is easy to implement.

NEWS-OS4.0 (operating system of SONY NEWS workstations, basically a 4.3BSD with NFS4.0) is used as an implementation workbench.

First, header files are modified to define new options as follows:

```
#define MNTOPT_DELAYIO   "delay"
#define M_DELAYIO        0x10000
#define VFS_DELAYIO      0x10000
```

along with small modifications of the code to propagate the new option flags appropriately from /etc/fstab to vfs structure.

Then, calls to sync or async write such as:

```
bwrite(bp);
```

are modified as follows:

```
if(ITOV(dp)->v_vfsp->vfs_flag &
   VFS_DELAYIO)
    bdwrite(bp); /* delayed write */
else
    bwrite(bp); /* sync write */
```

The kernel modifications was performed on 8 files in the sys directory: h/mount.h, h/vfs.h, h/vfs.c, ufs/ufs_bmap.c, ufs/ufs_dir.c, ufs/ufs_inode.c, ufs/ufs_vnodeops.c at 22 locations. The original total of 15 lines are modified into 49 lines.

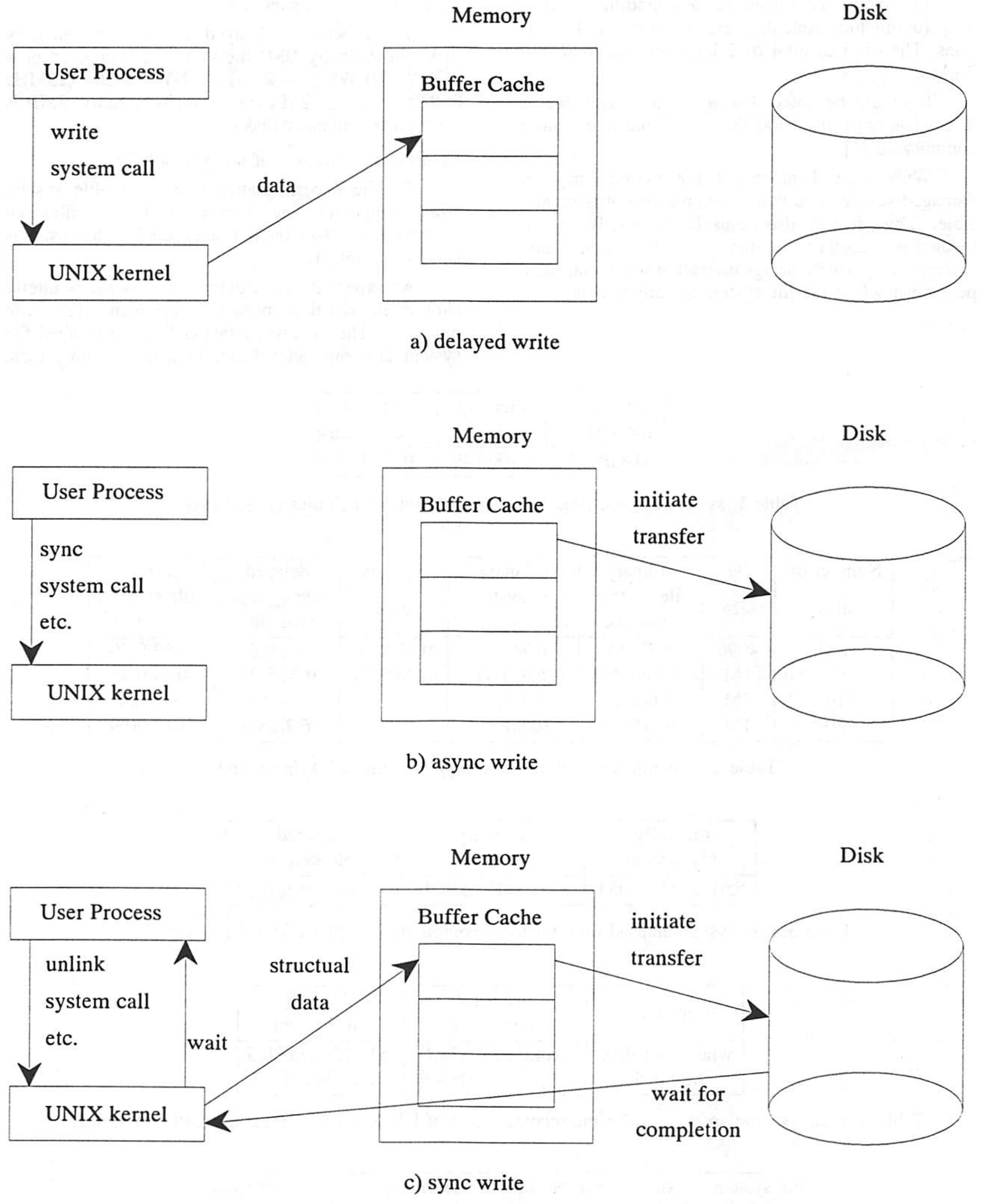


Figure 1. UNIX file system write strategies

The mount command is also modified over 2 files (usr/include/mntent.h, etc/mount.c) at 4 locations. The original total of 2 lines are modified into 7 lines.

It should be noted that, a memory disk device driver has more than 300 lines in C and much more complicated [1].

With delayed mount, a file system may be damaged severely, so that automatic reboot is impossible. Thus, it may also be useful to modify fsck to recognize another option in /etc/fstab, say, "autonewfs", which designate fsck to automatically perform newfs if the file system is unrepairable.

Performance Measurement

Speed between delayed file system, memory disk and ordinally BSD file system is compared on a SONY NEWS workstation NWS-1850 (25MHz 68030). Among 32M bytes of main memory, 8MB is used for the memory disk.

Creation and removal of many small files.

To check performance with many file system structure updates, time to create 100 empty files and then remove all of them is measured. The result is shown in Table 1.

As expected, the ordinary file system is utterly slow in elapsed time, because many many sync write is issued. The interesting fact is that the delayed file system is about twice faster than the memory disk.

ordinary file system	memory disk	delayed file system
0.42/8.45	0.48/0.49	0.25/0.28

Table 1. system/elapsed time to create and remove 100 files (in seconds)

Number of files	IO size	ordinary file system with dbc	ordinary file system without dbc	memory disk	delayed file system with dbc	delayed file system without dbc
100	4096	0.72/9.4	0.74/9.4	0.75/0.78	0.52/0.58	0.53/0.59
1	1M	0.39/1.40	0.38/1.41	0.76/0.76	0.34/0.35	0.34/0.35
10	1M	4.8/14.1	5.4/43.7	-	3.4/5.6	5.0/32.2
20	1M	10.9/28.4	10.6/87.8	-	6.7/25.9	10.0/70.9

Table 2. system/elapsed time for a large amount of IO (in seconds)

ordinally file system	memory disk	delayed file system
850.5/107.9/1055	848.5/119.85/1013	848.3/108.6/1024

Table 3. user/system/elapsed time for the recompilation of UNIX kernel (in seconds)

media	ordinally file system	delayed file system
winchester disk	514.0/87.4/872.1	514.1/84.7/680.5
MO disk	514.2/86.5/1804.9	513.6/86.2/676.6

Table 4. user/system/elapsed time for the recompilation of UNIX kernel with faster CPU (in seconds)

file system name	size (MB)	number of i-nodes	ordinally mount	delayed mount
/	3	573	0.1/2.5/47.8	0.1/1.9/7.0
/var	1	156	0.0/1.0/26.7	0.0/0.7/3.0
/usr	120	9293	4.9/116.7/1949.4	4.8/121.4/1442.3

Table 5. user/system/elapsed time for system installation (in seconds)

The memory disk is slower than the delayed file system and consumes more system time than the ordinally file system, because it requires copying of data from buffer cache to memory disk, which is performed by CPU.

Intensive IO on a large file.

To check performance with the combination of file system structure updates and data transfer, time to create several files, to write some amount of data, to read the data and to remove the files is measured. Measurement was done with and without dynamic buffer cache [4]. Without dynamic buffer caching, only small (5 to 10 %) amount of main memory is used as buffer cache. With dynamic buffer caching, all available memory can be used as buffer cache. The result is shown in Table 2.

As the size of IO becomes large, the difference between the ordinary file system and the delayed file system become smaller. This is because most of write operations are on plain data and thus already delayed write even without a delay option. On the delayed file system, IO operation is still faster than the memory disk. It is interesting that the memory disk consumes even more system CPU time than an ordinary file system, when IO size is 1MB. This is because, data transfer of an ordinary file system is performed by a DMA controller parallel to the CPU.

For a large amount of IO, where memory disk can not hold all of the data, the delayed file system is still usable and a little faster than the ordinally file system. Dynamic buffer caching is effective to allow fast IO of large amount of data.

Recompilation of UNIX kernel

As a problem in the real world, time to compile entire /vmunix from the source code is measured.

As shown in Table 3, the result is rather disappointing. No meaningful difference is observed because compilation is basically CPU bounded, that is, most of elapsed time is consumed by user CPU time.

So, we made another measurement with a faster CPU. This time, a SONY NEWS workstation NWS-3800 (20MHz R3000) is used. We also made a measurement with a MO (Magneto-Optical) disk mounted on /tmp, as the MO disk is much slower than winchester disks. The result is shown in Table 4. As expected, elapsed time of the compilation is reduced with the delayed /tmp file system. The result with the MO disk suggest the possibility to use something like delay option on slow IO such as an NFS mounted /tmp file system.

New installation of system

Delay option is also applicable to install a totally new file system. When a whole file system is installed, the file system contains nothing at first. So, if the installation fails due to a system crash and all the contents of the file system are lost, it is not a problem. Thus, by specifying **delay** option, faster installation is expected. So, we measure improvement with the actual installation tapes (1/4" QIC24, tar+compress format). The result is shown in Table 5.

The delay option is more effective for / and /var file system than /usr file system. This is because / and /var are small and all the content can be held in main memory.

Conclusion

By introducing a new mount option **delay**, it is possible to obtain a fast /tmp file system. The **delay** option is actually implemented and compared against an ordinally file system and a memory disk file system on implementation effort, speed, maximum file system size and volatility. The delayed file system is better than the memory disk in all respects.

References

1. der Mouse, "Silicon disk driver for BSD UNIX," *comp.sources.misc <2677@ncoast.UUCP>*, USENET, (jun. 1987).
2. D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," *Communications of the ACM* **17**(7) pp. 365-375 (Jul. 1974).
3. S.J. Leffler, M.K. McKusick, M.J. Karels, and J.S. Quarterman, *4.3BSD UNIX Operating System.* 1988.
4. R. Rodriguez, M. Koehler, L. Palmer, and R. Palmer, "A Dynamic UNIX Operating System," *Summer USENIX'88*, pp. 305-320 (Jun. 1988).

Masataka Ohta received B. S. in Computer Science 1982 and then M.S. in 1984 from the University of Tokyo. At the same time, he engaged in creating computer graphics systems and computer generated images at Life Structure Institute, Seibu Digital Communications and FROGS. He is currently a research associate at the computer center of Tokyo Institute of Technology and managing tens of UNIX systems. Ohta is a member of ACM and the Information Processing Society of Japan. His address is: Computer Center, Tokyo Institute of Technology; O-okayama, Meguro-ku; Tokyo 152, JAPAN. Reach him electronically at mohta@cc.titech.ac.jp .

Hiroshi Tezuka left University of Tokyo before graduation in 1980. Then, he joined Life Structure Institute in 1981. He began to develop UNIX workstation in SONY from 1985. He is currently a researcher of SONY Computer Science Laboratory, and interested in virtual memory sub-system and multi-processor operating system. His address is: Sony Computer Science Laboratory, Inc.; 3-14-13, Higashi-gotanda,; Shinagawa-ku, Tokyo 141, JAPAN. Reach him electronically at tezuka@csl.sony.co.jp .

USENIX Summer Conference
June 11-15, 1990
Anaheim, California

DEcorum File System Architectural Overview

Michael L. Kazar, Bruce W. Leverett, Owen T. Anderson, Vasilis Apostolides, Beth A. Bottos, Sailesh Chutani, Craig F. Everhart, W. Anthony Mason, Shu-Tsui Tu, Edward R. Zayas – Transarc Corp.

ABSTRACT

We describe the DEcorum file system, a distributed file system designed for high performance, low network load, easy operation and administration, and interoperability with other file systems. The DEcorum file system has three components: the DEcorum protocol exporter (or file server); the *Episode* physical file system; and the DEcorum client (or cache manager). Episode is a module that implements the Vnode/VFS interface, using transaction logging to allow fast recovery from crashes.

To be exact, it implements a *VFS+ interface*: extensions to the standard Vnode and VFS interfaces, and two new modules, *aggregates* and *volumes*, which give flexibility beyond what is provided by Unix partitions to support administration and operation of networks of thousands of workstations. The DEcorum protocol exporter provides remote access to the Episode physical file system via remote procedure calls (RPC's). It can export access to other physical file systems, such as the Berkeley fast file system, using extensions of the physical file systems to support the VFS+ interface.

The DEcorum client exports a Vnode interface, but obtains its data by making RPC's to a DEcorum protocol exporter. It caches data from the file server. To synchronize accesses to files, preserving single-system UNIX semantics, it relies on typed *tokens* obtained with the data: guarantees provided by the server that various operations can be performed remotely. Tokens can be revoked by the file server using separate RPC's. A locking hierarchy is used to avoid deadlock between clients accessing files and servers revoking tokens on the same files; we explain the hierarchy and informally sketch a proof of its correctness.

History and Motivation

The DEcorum file system is a component of a larger system, the DEcorum Distributed Computing Environment (DCE). The DEcorum architecture was jointly developed by Hewlett-Packard, IBM, Locus Computing, and Transarc. The DEcorum file system component is a follow-on of AFS (formerly the Andrew file system).[1] The DCE is built as a layered architecture from modular components. The file system utilizes a collection of modular components, including Hewlett-Packard's NCS 2.0[2] remote procedure call facility, Hewlett-Packard's PasswdEtc authorization component, MIT's Kerberos authentication system, and many others.

AFS is a distributed file system originally developed at Carnegie Mellon University's Information Technology Center (ITC); its development has subsequently been taken over by Transarc Corporation. It offers local-remote file access transparency, and ease of operation and administration. It uses caching to achieve high performance and low network load.

The goal of the design of the DEcorum file system is to carry over the benefits of the existing AFS design, but to improve on it in its known areas of weakness. Specifically,

- Interoperability with existing file systems is improved, so that if a file server is installed on a host running UNIX, the server can export file systems that were already in use on that host.
- File server availability is increased (mean time to recover is decreased), through design of a physical file system that does not require a lengthy file system salvage process after a crash.

[1] Alfred Z. Spector and Michael L. Kazar, "Uniting File Systems," *Unix Review*, March 1989.

[2] NCS 2.0 adds support for pipes (streaming), long-haul operation, authentication, and connection-oriented transport to NCS 1.5 (Lisa Zahn *et al.*, *Network Computing Architecture*, Prentice Hall, 1990).

- Improved caching algorithms support strict UNIX single-system semantics in sharing of files. As a side benefit, network traffic is lessened, and client performance improved.

This list is not meant to be exhaustive. Some other ways in which improvements are made to AFS are mentioned in the following section.

This paper describes the design of the DEcorum file system. We emphasize the aspects that improve upon the older AFS design. To provide context, we also describe some aspects that were carried over from AFS.

Overview

The three principal improvements described in the previous section are achieved as follows:

- Interoperability with existing file systems is improved by a clean separation between the higher level of the file server, called the *protocol exporter*, and the lower level, which is the *physical file system*. The separation is at the level of the virtual file system (VFS).[3] We define a physical file system as a module that implements the VFS interface, and stores file data on a disk (as opposed to using file data retrieved from other nodes in a network). Though a physical file system has been designed specifically for use with the DEcorum protocol exporter, other physical file systems can be used. For instance, the file systems, descended from the Berkeley Fast File System,[4] supplied by vendors of various platforms can be exported. Section 1 describes the structure of the DEcorum client and file server, and their interfaces with non-

[3] S.R. Kleiman, "Vnodes: An Architecture for Multiple File System Types in Sun UNIX," *USENIX Conference Proceedings* (Atlanta, Summer 1986), pages 238-247.

[4] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry, "A Fast File System for UNIX," *ACM Transactions on Computer Systems*, Vol. 2, No. 3, August 1984, pages 181-197.

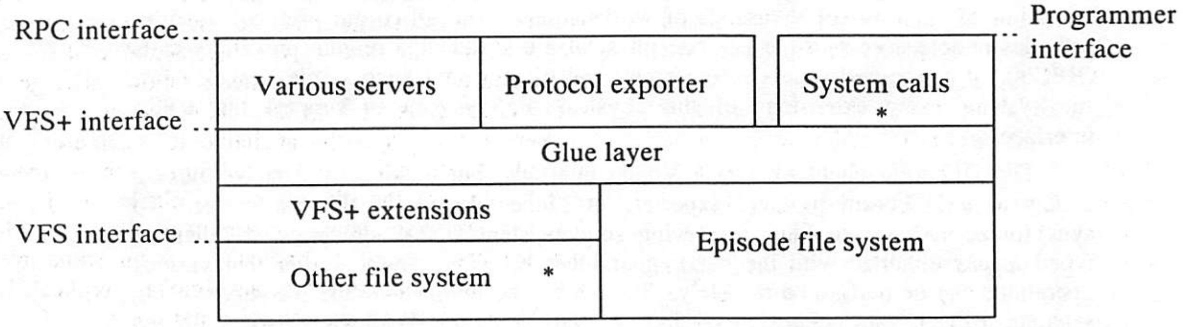


Figure 1: File server organization

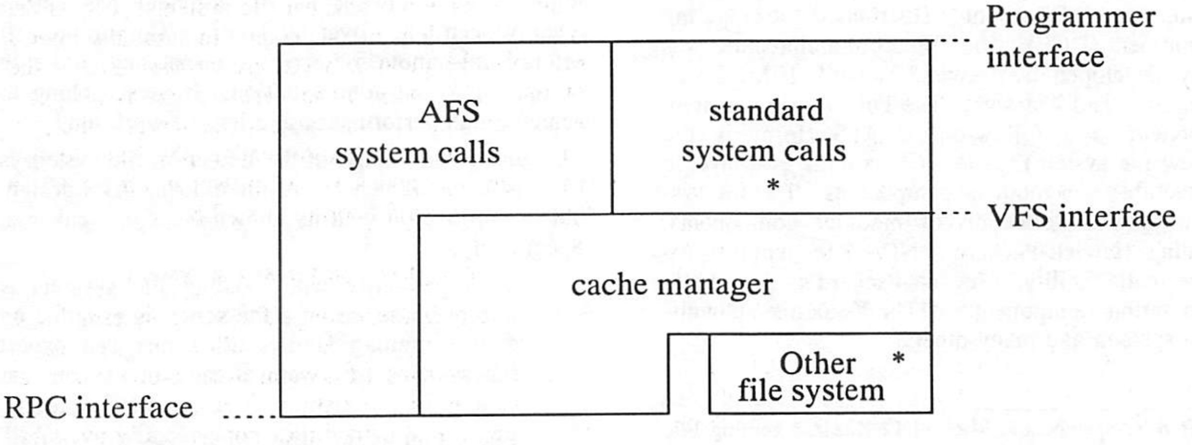


Figure 2: Client organization

DEcorum file systems and other code. In Sections 2, 3, and 4 we expand upon the structural descriptions of Section 1: Section 2 describes the Episode file system, Section 3 describes the DEcorum protocol exporter and other server components, and Section 4 describes the DEcorum client.

- Cache consistency is achieved by the use of typed *tokens*, representing guarantees made by the file server to the client about what operations the client can perform locally. The corresponding ''callback'' system in AFS, because it was untyped, was not capable of representing the full variety of guarantees that might be useful to the client. Section 5 describes the token system. Section 6 discusses the potential deadlocks that arise from caching, and how they are avoided in the context of the token system.
- The need for a file system salvage (the notorious *fsck*) is obviated by the use of *logging* in the server's physical file system. Section 2.2 describes the logging and recovery system and its position in the structure of Episode. We show that logging, far from causing excess disk activity during routine operation, actually *improves* performance, by reducing the need for synchronous writing of meta-data to disk.

Other differences between DEcorum and AFS are described incidentally to various sections. Particularly worthy of note are POSIX-compliant access control lists (Section 2.3), provision for diskless clients (Section 4.2), and provision for lazy replication of files (Section 3.8).

Throughout the paper we compare the DEcorum design with those of other distributed file systems, notably AFS and NFS,[5] and with non-distributed file systems. In the last section we summarize these comparisons and summarize the contribution of the DEcorum design.

System structure

1. Overview of system structure

We speak of a node in a network as either a *file server*, maintaining file systems on disk and *exporting* them to other nodes; or a *client*, running applications that access files exported by servers. A node can simultaneously act as both server and client. The DEcorum file system includes software for both servers and clients. Figure 1 shows the structure of the server side of a DEcorum file system, and figure 2 shows the structure of the client side. In both diagrams, an asterisk indicates a component that is not part of the DEcorum software, that is, a component taken from whatever existing UNIX kernel runs on the node.

Client and server communicate via remote procedure calls (RPC's); thus RPC interfaces are shown in both diagrams. On the client side, the community of server file systems appears as a single file system, presenting a Virtual File System (VFS) interface to the Unix kernel.[6] The *cache manager* is the component that implements the remote file system, including subcomponents to manage a cache (using a native file system), and to fetch and store files from and to remote servers.

On the server side, we again rely on the VFS interface. A file system that is to be exported must present a VFS interface, to which we add (to the extent that it is practicable) extensions, called the *VFS+ interface*, to make possible the volume-level operations discussed in later sections. A *glue layer* implements synchronization, ensuring that each VFS+ operation acts on files on which sufficient locks (*tokens*) are held to guarantee serializable results. The glue layer is transparent from the point of view of the programmer, in that it presents the same VFS+ interface to the layers above it as is presented to it by the file systems below. The generic system calls sit atop the VFS+ and glue layers, making the file systems available to local users of the file server node; but in addition, the *protocol exporter* makes the file systems available to clients. The diagram also shows a component labeled ''various servers;'' these components run on some file servers, and maintain various (global, replicated) databases for use by client cache managers.

The client structure of Figure 2 is carried over with few changes from AFS. The server structure of Figure 1, however, has changed. In an AFS server, the protocol exporter and underlying file system were not separate components. Thus the protocol exporter could not make use of other file systems, such as the native file system of the node. In turn, the node's own kernel could not make use of the server's file system. (Only if the node were a client as well as a server could local users access the server's file system.) The ability to export native physical file systems to client nodes is the principal difference in function between DEcorum and AFS.

2. Episode physical file system

Episode is a fast-restarting UNIX file system. It is intended to provide the capabilities needed for a large-scale distributed system with performance as good as the best existing physical file systems. There are several capabilities not generally available in vendor-supplied file systems. Most important are

[5] E. Walsh *et al.*, ''Overview of the Sun Network File System,'' *USENIX Conference Proceedings* (Dallas, Winter 1985).

[6] S.R. Kleiman, *op. cit.* The interface between generic system calls and specific file systems varies between different vendors' UNIX kernels; in this paper, for simplicity, we use the name VFS for all of them.

support for logical volumes, support for access control lists, and the ability to recover quickly from crashes.

We assume that Episode is based on a standard UNIX disk partition using the facilities of the kernel device driver. It supports both local and remote use: the former as a local file system when individual volumes are mounted; the latter as exported by the DEcorum protocol exporter. To provide a uniform level of access, it implements an upward compatible superset of the standard VFS functions.

The code is designed to take advantage of a multi-threaded environment and asynchronous I/O. Thus it would normally run in the UNIX kernel, although a non-kernel environment with those facilities would not be precluded. Running in the kernel also eliminates the system call overhead paid by mixed kernel and user implementations, as observed in the AFS file server.

Because Episode is intended to support high levels of concurrent access, it is designed with finely grained locking, and as few points of global contention as possible.

In the next sections we describe the capabilities of Episode that represent advances over older physical file systems: the volume/aggregate concept; access control lists; and logging. We also discuss some implementation issues, including the *anode* abstraction, which is comparable but not equivalent to the inode structure in other systems.

2.1 Volumes and Aggregates

In many UNIX environments, a file system (a mountable subtree of the directory hierarchy) is identified with a partition (a disk, or a quantity of disk storage, managed as a unit). In AFS, and in DEcorum, these two concepts are distinguished. To avoid conflict with standard UNIX terminology, a mountable subtree is called a *volume*, and a unit of disk storage is called an *aggregate*.

The volume/aggregate distinction is carried over to DEcorum from AFS.[7] We review it in this paper, however, because it is so important. Administration of networks of thousands of users is not practical without this distinction.

The distinction allows volumes to be *moved* among partitions, and even moved from one server to another. This capability in turn allows the system administrator to perform load balancing. Episode supports dynamic volume motion, without taking down any servers or making any facilities unavailable except the volume itself, and that transparently to application programs (which are blocked for a short time).

Equally importantly, volumes can be *cloned*, that is, a read-only copy (snapshot) of a volume may be made within an aggregate. Using volume cloning and moving, the system administrator can improve load balancing and availability of utilities. The administrative procedure of cloning a volume and moving the clone(s) is called *read-only replication* of the volume; we discuss replication further in Section 3.8. Episode supports cloning at the lowest level: a *copy-on-write duplicate* of a file can be created, in which, instead of data blocks and indirect blocks, there are pointers to the corresponding blocks of the original. The original file becomes the read-only version, while in the copy, any writing of data causes separate copies to be made of just as many blocks as required.

The unit that is backed up is the volume, not the aggregate. Thus the system administrator can back up a volume by cloning it, and later (at leisure) writing the clone to removable media. The volume is unavailable only transparently, for the time required by the cloning; and only one volume is unavailable at a time. Moreover, the clone can continue to exist on disk indefinitely; while it exists, files can be restored from it directly, bypassing the removable media.

In addition to the standard Vnode and VFS interfaces, Episode implements a volume interface and an aggregate interface. A VFS is a mounted volume, but the volume interface is separate from the VFS interface, because operations such as moving and cloning can be performed on volumes that are not mounted.

2.2 Buffer package and logging system

Crash recovery in the DEcorum file system is based on using log-based recovery techniques to either undo the operations that were begun but not finished, or to complete operations that finished, but had not been completely written to the disk.

The goal of using logging techniques for this purpose is to enable the DEcorum file system to restart operation as quickly as possible after a crash. Certain operations, if interrupted, would leave the file system in an inconsistent state, unsafe to use. Logging and the associated recovery techniques will allow the system either to complete or to completely undo such operations before resuming normal operation.

It should not be expected that logging would entirely eliminate the need for disk salvaging. Media failure will normally necessitate salvaging.

One might expect that logging would reduce overall file system performance due to the fact that certain information must be written both to the file system itself and to the log. Instead we expect the performance of the DEcorum physical file system to exceed that of the Berkeley fast file system (FFS).

[7] John H. Howard *et al.*, "Scale and Performance in a Distributed File System," *ACM Transactions on Computer Systems*, Vol. 6, No. 1, February 1988, pages 51-81.

The FFS schedules large numbers of writes to file system meta-data as soon as the meta-data are modified. Examples of such meta-data include inodes, indirect blocks and directories. While these updates are generally performed asynchronously, they generate so much disk traffic that overall system performance is affected. In addition, a second update to a disk block already queued for one of these asynchronous writes will generally wait until the write operation completes, also hurting performance. The FFS performs all of these extra writes in order to ensure that certain information is written before other information, to simplify the job of *fsck*.

In comparison, a log-based file system need not force modified meta-data to the disk in order to guarantee file system consistency in the event of a crash. Rather, the file system will be consistent, if a bit old, after the log has been replayed. The file system may periodically batch-commit all pending transactions, but fidelity to the spirit of the UNIX file system only requires batching commits every 30 seconds, or when a *sync* or *fsync* system call is executed. In addition, these batch commits only require writing data sequentially to the end of the log; disks are especially efficient at performing these types of writes.

Thus, the DEcorum file system should actually generate considerably fewer disk updates, especially when performing operations that primarily change file system meta-data, such as file creation, deletion, and truncation.

Each aggregate has a log. This is an area of disk, not necessarily contiguous, whose size is fixed at aggregate initialization. The log is not necessarily located on the aggregate that it is logging.

Changes to meta-data are logged; changes to user data are not. Changes are grouped into *transactions*, defined in the usual way: each transaction must be atomic, that is, no individual change will take effect unless all the transaction's changes do. The log entry for a change gives the old and new values for all data bytes in the change, and the identity of the transaction of which the change is part. A separate log entry notes when a transaction *commits*, that is, when all the changes comprising it have been logged. The recovery procedure replays the log, completing transactions that had committed, and undoing transactions that did not commit. The time spent in recovery is proportional to the size of the active portion of the log, not (as with *fsck*) to the size of the file system.

Transactions may not span across calls to the VFS interface, but one such call may be comprised of more than one transaction. In particular, long operations are broken into sequences of short-lived transactions, each of which leaves the file system in a consistent state. For example, truncation of a file may be broken up to truncate only one block or a few blocks at a time. Breaking up operations enables us to guarantee that transactions will be short-lived, which in turn enables us to keep the log to a small, fixed size without requiring complex algorithms for log truncation.

Transactions must be made *serializable*. Suppose, for instance, that transaction A is ready to commit, after using data modified by transaction B, which is not yet ready to commit. It must be ensured that A will not commit unless B commits also. Our implementation of serialization is outside the scope of this paper.

The logging system is intricately entwined with the disk buffer cache. Higher-level file system functions must not modify buffer data directly, but instead go through logging primitives. Moreover, the same higher-level functions, having modified a buffer, do not distinguish between different degrees of synchrony in writing the buffer; they simply release it, leaving the writing to the logging system. With each buffer, the logger records the position of the most recent log entry for changes to the buffer's data; the buffer must not be written to disk until the log has been flushed to disk up to that position.

2.3 Access Control Lists

The idea that the mode bits associated with each file in UNIX do not allow enough flexibility in authorization strategies is not a new one. Several UNIX file systems, including AFS, supplement the mode bits with *access control lists* (*ACL*'s), association lists in which users or groups are paired with specific sets of rights to be added or denied in accessing a file. A complete description of the semantics of DEcorum ACL's is outside the scope of this paper. The principal difference between DEcorum and AFS ACL's is that in DEcorum any file or directory may have an ACL, while in AFS only a directory may have one.

2.4 Anodes

From a user programming point of view, perhaps the most important aspect of the abstraction offered by a file is its open-endedness: one can write to the file to any length, without worrying about where the storage for it is going to come from. In many UNIX file systems, this open-endedness is implemented at the level of the inode. Other aspects of the abstraction are implemented at the same level: authorization (mode bits and ACL's), status information (such as access and update times), and, not least, the directory hierarchy (and link counts). The ''bundling'' of all these abstractions is inconvenient to the kernel programmer, for it discourages him from taking advantage of open-endedness in situations where he does not want the other abstractions. He must either abandon open-endedness in favor of fixed limits, or re-implement some form of open-ended resource usage himself. An example of the former strategy is the fixed size limit on ACL's in AFS; readers familiar with other UNIX file systems can no doubt think of other examples of both strategies.

The DEcorum *anode* abstraction provides an open-ended address space of disk storage *and nothing more.* (To use more exact terminology, the actual disk storage is called a *container*, and the anode is the small set of bytes that serves as a descriptor for it, either on disk or in memory.) Anything that uses storage on disk is implemented as an anode: files, ACL's, volumes, aggregates, transaction logs, disk allocation bitmaps, and so on. Files are implemented as anodes with additional bells and whistles: a set of status bytes, a pointer to an ACL, and a position in the directory hierarchy.

Because all data and meta-data are stored in anodes, the disk presents a uniform interface to utilities that access it, even if they access it below the file system level. For instance, the logging system and the salvager are somewhat simpler than they would be if they had to distinguish between anode and "other" disk areas.

3. DEcorum protocol exporter and related file system servers

We describe the individual components of a DEcorum protocol exporter. We also discuss some of the global database servers that play roles in the DEcorum file system, although these are not components of file servers. Some of this structure is carried over unchanged from AFS, and so we describe it only briefly.

3.1 Token manager

The token manager maintains, for each file in use, a set of guarantees that have been made to various clients. These guarantees have a binary *compatibility* relation telling whether two guarantees can be simultaneously granted to two separate clients.

3.2 Host model

The host model maintains structures describing authenticated individuals that have made RPC's to it, and the client managers from which the RPC's originated. Information about the state of the client includes, for instance, whether all of the token revocation messages issued to it have been delivered. Information about an individual might include the Kerberos[8] (or other authentication system) identity of the caller.

3.3 Vnode glue layer and VFS+ interface

The purpose of the glue layer is to synchronize actions on files. For each Vnode operation provided by a conventional file system, a corresponding "wrapper" operation is substituted that obtains tokens and then performs the original operation. This layer must avoid deadlock; in this sense it faces the same problem, in perhaps aggravated form, that physical file systems face. For some operations, directory lookups must be performed to complete the set of Vnodes for which tokens must be obtained before the physical file system operation can be called.

In addition to the usual VFS and Vnode operations, the protocol exporter allows for additional operations to provide access to such extensions as volumes and access control lists. The Episode physical file system, of course, implements all these operations. For any other physical file system, it may be possible to provide some subset of DEcorum functionality by writing the corresponding operations. For instance, some volume operations could be implemented in a conventional UNIX kernel, assuming a mapping between volumes and file systems and between aggregates and partitions, even though no more than one file system can occupy a partition.

3.4 Volume registry (and volume location database)

The volume registry is a simple symbol table enumerating the set of volumes residing locally on the server. It can be used within the server, whereas the *volume location database*, a global replicated database describing which volumes are on which servers, provides service to remote clients.

3.5 Server procedures

The server procedures implement the RPC interface in terms of calls to the previous components.

3.6 Volume server

The volume server implements per-volume operations, such as moving a volume from one file server to another, and makes them available to administrators at remote clients.

3.7 Authentication server

All RPC's are authenticated. The DEcorum authentication service is based on Kerberos. A description of it is outside the scope of this paper.

3.8 Replication server

In AFS, the benefits of read-only replication of a volume, as described in Section 2.1, become available only when a system administrator or operator explicitly requests creation of a replica. The DEcorum replication service implements *lazy replication* of volumes: a replica is maintained permanently, and is guaranteed to be out of date by no more than a fixed amount of time. In principle, as the amount of time approaches zero, lazy replication approaches instant (i.e. read-write) replication; but in practice, the design of the lazy replication system is not expected to perform well under those circumstances, e.g. when the amount of time is less

[8] J.G. Steiner, C. Neuman, and J.I. Schiller, "Kerberos: An Authentication Service for Open Network Systems," *USENIX Conference Proceedings* (Dallas, Winter 1988).

than about 10 minutes. The client of the replica is guaranteed to always see a consistent snapshot of the volume, and is guaranteed that data in the replica are never replaced by older data. A replication server requests a *whole-volume token* to guarantee that it can use a replica of a volume; when it must update the replica, it attempts to obtain from the master copy only those files that have changed.

4. DEcorum client

We describe the four layers into which a DEcorum client can be divided. As with the DEcorum protocol exporter, much of the structure is carried over with little change from AFS, and so is described here only briefly.

4.1 Resource layer

The lowest layer is the resource layer. It maintains RPC connections and caches volume location information. When upper-level modules desire an RPC connection authenticated as a particular user, it is this module that they call.

4.2 Cache layer

The cache layer caches status and data information from files stored at remote file servers. It also checks that cached information is up to date. In AFS clients, vnode status information is cached in memory, while file data are cached in disk files provided by the ''native'' (generally vendor-supplied) physical file system. This structure is carried over to DEcorum, with the exception that an in-memory version of the data cache is provided as an option, enabling diskless clients to be used. The cache layer uses tokens to ensure consistency, as described in detail in Section 5.

4.3 Directory layer

The client has its own directory module. This enables it to perform lookups without contacting the server, assuming that it has up-to-date directory data to use. In AFS, the client copied a directory from the server on the first reference; subsequent directory operations could be performed on the client using exactly the same code used in the server. In DEcorum, this technique is not always available, since in general the client will not understand the directory format used in every server file system. Instead of caching whole directories, the client must in general cache the results of individual lookups.

4.4 Vnode module

The highest layer, the *vnode layer*, implements the Vnode and VFS interfaces required for the client's kernel, in terms of calls to the three lower layers described above.

Issues

5. Caching and tokens

5.1 Tokens

The synchronization challenge in the DEcorum file system is to ensure that all users of a physical file system exported by the DEcorum protocol exporter receive single-system UNIX file access semantics, no matter whether they reference the data remotely from a client or locally via a local mounted file system. Remote clients must read and write file status and data, but ought not to incur the expense of making an RPC to the file's home machine in order to call the appropriate Vnode operation directly for every remote reference. What is required is a mechanism for allowing operations to be performed on the client, while simultaneously blocking conflicting operations at other sites. Conflicting operations, in this sense, are those that, if performed, would cause incorrect results to be obtained by other clients or local users.

The architecture chosen to solve this problem is simple. Each server includes a *token manager*, which keeps track of who is referencing files, what they are doing to the files, and what guarantees they require about what others may do to the files. For example, a protocol exporter may allow a client to read the contents of a file (from the client's own cache) until otherwise notified. The protocol exporter records that the client has received a guarantee, and it will not allow anyone to write data to the file without first revoking that guarantee (i.e. notifying the client that its cached data must no longer be used).

The token manager is invoked by *all* calls through the Vnode interface, and ensures that any guarantees incompatible with the operation being attempted are first invalidated. The token manager is invoked by the Vnode layer because non-DEcorum protocol exporters (such as NFS), as well as locally executed system calls, may perform various operations on the resident physical file systems, and these operations must be synchronized with the guarantees exported by the DEcorum protocol exporter.

We use the term *client* in this section to denote any entity, remote or local, that requests tokens on the files it desires to work with. This terminology conflicts with our usual definition of *client* as a remote user of files exported by a file server, but it is the conventional way of describing the relation of module users to a module. There are many potential clients of a token manager, including local UNIX kernels and remote file system protocol exporters. A client of a token manager registers itself with the token manager in a fairly general manner: it passes in an object of type *afs_host*, having a virtual *revoke* procedure. The *revoke* procedure is called whenever the token manager needs to revoke the token.

5.2 Token types

Some tokens are incompatible with other tokens. Before granting a new token, the token manager may have to revoke some already-granted tokens. The tokens currently defined provide the following guarantees:

- **Data tokens** grant the holder the right to read or write (depending on the subtype of the token) a range of bytes in a file. A read data token allows the possessor to cache and use a copy of the relevant file data without repeatedly performing RPC's to the appropriate file server, either for validating the data or for re-reading it. A write data token allows the holder to update the data in a cached copy of the file without storing the data back to the server or even notifying the server.
- **Status tokens** allow the holder to read or write (depending on the subtype of the token) the status information associated with a file. A read status token allows the holder to refer to cached copies of the status information without calling the server to check the status. A write status token allows the holder to update its cached copy of the file's status without notifying the server. The token manager blocks other VNODE functions from even looking at the file's status information before revoking any write status tokens.
- **Lock tokens** allow the holder to set read or write file locks (depending on the subtype of the token) on a particular range of bytes within a file. Without holding a lock token, a client must call the server to set a file lock; with such a token, the client is assured that the server will not attempt to set conflicting locks on the file, without first revoking the token.
- **Open tokens** grant the holder the right to open a file. There are different subtypes for different open modes: normal reading, normal writing, executing, shared reading, and exclusive writing.

Tokens of any type are compatible with tokens of any other type, as they refer to separate components of files. Tokens of the same type may be incompatible with each other:

- Read and write data tokens are incompatible if their byte ranges overlap.
- Read and write status tokens are incompatible.
- Read and write lock tokens are incompatible if their byte ranges overlap.
- The compatibility matrix of open tokens is given in Figure 3.

Tokens are managed in the DEcorum file system by modifying all the Vnode functions to first call the token manager to obtain the appropriate tokens for whatever operations they will perform, then to perform the operations, and finally to call the token manager to notify it that the tokens can now be revoked at any convenient time. The modified Vnode functions constitute the *glue layer.*

open for normal reading	C	C	C	I
open for normal writing	C	C	I	I
open for shared reading (= open for executing)	C	I	C	I
open for exclusive writing	I	I	I	I

Figure 3: Compatibility matrix of open tokens
(C = compatible, I = incompatible)

5.3 Token revocation

When the token manager wishes to revoke a token, it notifies the client to which the token was granted. If the token is for reading (status or data), it must simply be returned. If the token is for writing, the client must write back any status or data that it has modified, before returning the token. If the token is for locking or opening, the client may elect not to return the token at all; this is the normal action if the client has already locked or opened the file.

For remote clients, revocation notifications are sent by RPC's. Thus RPC communication between DEcorum clients and DEcorum servers is two-way: clients call servers to access files, and servers call clients to revoke tokens. This structure was not fully depicted in figures 1 and 2, in order to simplify those diagrams.

5.4 Comparison with other distributed file systems

Tokens implement the strongest possible consistency guarantee for users of a shared file: when one user modifies a file, other users see the modifications as soon as the *write* system call is complete. At the same time, communication between client and server is kept to the minimum, that is, shared data are transmitted only when they are actually shared. In this section we compare tokens with the spectrum of distributed file system semantic models and implementations described by Kazar.[9]

Relatively weak cache consistency guarantees are provided by the Sun Network File System (NFS). A page of cached file data is assumed to be valid for 3 seconds; if it is directory data, it is assumed to be valid for 30 seconds. The use of fixed time limits hampers the writer of a distributed application; the application must wait the specified length of time before fetching shared data, if it relies on the correctness of the data. This disadvantage of weak consistency guarantees is not accompanied by a

[9] Michael Leon Kazar, "Synchronization and Caching Issues in the Andrew File System," *USENIX Conference Proceedings* (Dallas, Winter 1988), pages 31-43.

corresponding advantage, such as low network utilization; clients must communicate with servers every 3 seconds whether or not any shared data have been modified.

AFS provides consistency guarantees at an intermediate level of strength. AFS "callbacks" are roughly equivalent to DEcorum status read tokens, in that the operations for which the AFS client would get a callback are those for which the DEcorum client would request a status read token. But because callbacks are the only synchronization mechanism, they are overburdened. There are not separate callbacks for reading and writing, nor for status and data. Thus the AFS client cannot know when to store back to the server any data that it has modified in the cache.

In the absence of certain knowledge, the AFS client could preserve single-system semantics only by communicating with the server at every *write* system call. (The server could then break callbacks to other clients reading the data.) Instead, it stores data back to the server when the file is closed.

This system has, in perhaps diminished form, the same drawbacks as that used by NFS. The consistency guarantees are not strong enough for some distributed applications, yet communication with the server is not minimized (i.e. there is communication at every *close* system call). The use of typed tokens allows DEcorum to synchronize access to files accurately but with minimal communication.

Two further limitations of the AFS callback system should be noted:

- Callbacks cannot describe byte ranges of data. If a group of users are accessing (and modifying) the same large file, even though they may be using disjoint parts of it, the file will frequently be shipped back and forth in its entirety between nodes.
- Callbacks cannot describe open modes. Any number of AFS users may have a file open in any mode at a time. For clients implementing some versions of the UNIX file system, this is not a pressing problem, inasmuch as UNIX allows multiple users to open the same file for reading or writing simultaneously. But the exotic open modes enable clients to implement the semantics of non-standard or even non-UNIX file systems. In addition, the UNIX restriction against opening a file for writing if it has been opened for execution can be implemented; and a virtual file system can assure itself that a file about to be deleted has no remote users, by requesting an open token for exclusive writing on the file.

5.5 Example

We describe a short synchronization example, showing the roles of the token manager and protocol exporters in synchronizing access to a file. Consider a file stored in a BSD UNIX physical file system, being written to by both a local user, who is issuing both read and write system calls, and a remote user, who is doing the same thing to the file via a client cache manager.

We begin with the remote application issuing a *write* system call to the file. This operation is handled by the client's cache manager, which requires a guarantee that it is permitted to locally update the file. This guarantee is requested from the protocol exporter of the server node on which the master copy of the file resides. The protocol exporter registers the client with the server node's token manager as having a data write token. Once the client receives the data write token, it can handle all remote writes to the file in question without contacting the master copy.

Assume that at some point a process on the server, accessing the file locally (not through a DEcorum client), decides to write some data to the master copy of the file. The process calls VOP_RDWR (the Vnode layer call for performing reads and writes) on the local Vnode, and the Vnode glue code first calls the local token manager, requesting a write data token for the file. Since there is a conflicting write data token granted to the remote client by the DEcorum protocol exporter, this incompatible token must be revoked before the local process can be granted its own token. The token manager does this: the protocol exporter is invoked again, and as part of its revocation procedure, it makes an RPC back to the client cache manager, asking it to return its token. As a side-effect of returning the token, the client will also store back the modified data pages. Once the remote client has stopped using and has returned its guarantee, the protocol exporter can return from the revocation call made by the token manager. Once the protocol exporter has returned its incompatible write token, the new write data token can be granted to the Vnode glue code, which will then perform the actual data write operation (by calling the original virtual file system's VOP_RDWR function).

The Vnode glue code need not hold onto its write data token for very long; it can return the token any time after the VOP_RDWR call has completed execution. By contrast, remote clients hold onto tokens as long as they can, to avoid unnecessary RPC's.

Should the remote user issue another call to VOP_RDWR, the client will again have to contact the protocol exporter to get another guarantee, and the protocol exporter will have to call the local token manager again to obtain another data write token.

6. Deadlock analysis

A significant challenge in building a system as complex as the DEcorum file system is ensuring that the overall system is deadlock-free. There are two general ways of dealing with deadlock in a distributed system: deadlock *avoidance* by partial ordering of locks, and deadlock *detection* by detecting cycles in the *waits-for* relation. We have chosen deadlock avoidance in our design. We implement this by establishing a partial order of locked resources.

The next section describes the structure of the basic calls from the clients to the server and the revocation calls from the server back to the clients. We pay particular attention to the position in the locking hierarchy of any resources that are locked. It is important that operations are serialized at the server, and that it is clients who must reconstruct the server's serialization, and not the other way around.

The following section describes how a correctly working cache manager can be built, given the constraints of locking objects in the order required by the locking hierarchy.

6.1 Basic Structure

The trickiest subsystem to deal with in these terms is the file server / cache manager subsystem, since the client makes calls to the file server to perform operations, which may make calls back to other clients. These return calls are the stuff of which dependency cycles are made.

The resources of interest are the vnodes on the client cache manager and the vnodes on the file server. A vnode and the token state associated with it may in some cases be components of the same locked resource; we will ignore token state in the description below.

Consider the following straightforward design: when a client cache manager makes a call to a file server, it locks its own cache vnode, and then makes the RPC, causing the server to lock its vnode and make calls back to other clients, which will lock the vnodes corresponding to the same file at their own sites only for the duration of the return call. The file server will then complete its call, and release its locks (and thread), and finally the client that initiated the call will release its locks.

This approach fails because of the difficulty in ordering the client's vnodes with respect to the file server's vnodes. Suppose that both clients A and B decide to call server S to perform some operation on some file *V*. Client A first locks its local vnode for *V*, V_a, while simultaneously, client B locks its vnode, V_b. Both A and B then make calls to S, and one of them, say A, locks V_s first. If the processing of this request requires the revocation of a token from the other client (B in this example), then the request from A will require waiting for B to release the lock on V_b, while B is waiting for the lock on V_s, which results in a deadlock.

The general solution to this problem we have chosen in the DEcorum file system design is to provide two locks per client vnode. One lock serializes the high-level operations performed by the client, so that two separate threads on the same client do not attempt to fetch the same data pages for the same file simultaneously. A lower-level lock serializes operations on the client vnodes that may leave the actual vnode in an inconsistent state, such as having a file length inconsistent with the number of bytes written to the file. The higher level lock is locked by high level operations initiated by the client cache manager, and is not unlocked until the operation completes. The lower-level lock is generally held from the time a client operation begins to deal with a vnode until the time that the client makes an RPC to the server, but is released just before the server is called. As soon as the response is received, the client re-obtains the lower-level lock (never having relinquished the higher-level lock), and processes any revocation operations that executed concurrently with the RPC.

Thus, all operations lock resources in this order: first, the client initiating the operation locks the high-level lock on its local vnode. Next the client may make an RPC to a server, which will lock the vnode present at the server. The server, while holding its vnode lock on the physical file system's vnode, may make token revocation calls to other clients, which will lock the low-level locks on their respective vnodes. Thus one always locks high-level vnode locks first, then server vnodes, and then low-level vnode locks.

6.2 Correctness

To be convinced this locking architecture is viable, one must be convinced that the client and server can serialize operations in the same order, even though the server is really the process that determines the serialization order, by its locking of the server vnode V_s. One must be convinced that the cache manager's vnode state is consistent with the server's vnode state after the execution of any of these RPCs, no matter which concurrent revocation service messages were processed during the client-to-server RPC.

To aid in performing this serialization-after-the-fact, the file server marks every reference to a file with a time stamp, whether the reference is for a token revocation call initiated by the server, a token-returning call initiated by a client, or a call that simply temporarily locks a file but does not return any synchronization tokens to the client, but may return other information, such as the current file status. If operation O_x on a file is serialized at the file server before operation O_y then the per-file time stamp returned by O_x for that file will be less than the the time stamp for that file returned by O_y. Because the time stamp is per-file, operations such as *rename*, which affect multiple files, will return multiple time stamps, one for each file of interest.

Time stamps must appear in return parameters from calls that read or write status information at the file servers, as well as in token revocation calls initiated by the file server. The time stamp is used to serialize client operations after obtaining the lower-level lock after an RPC call returns.

6.3 Examples

Consider the concurrent execution of a call M_s to retrieve the status of a file F, which grants data read and status read tokens for F, with the execution of a token revocation message M_{tr} pertaining to F.

M_{tr} contains a time stamp indicating its serialization order with respect to F. It also specifies the token T to revoke by means of a unique token ID. Assuming that M_s is the call that returned T, the problem is that M_{tr} and the reply to M_s may be processed out of order, that is, when the client processes M_{tr}, it may not yet recognize T. Thus, if M_{tr} specifies a token that is not recorded as part of F's token state, and vnode state indicates that a call that may return a token is in progress, the client also queues the token revocation for later processing. When the in-progress RPC completes, the client obtains the low-level lock, and then serializes the token returning operation implicit in the reply with the queued token revocation operation. This ordering is done strictly on the basis of the per-file serialization counter.

A more complex example occurs when one page is being stored back by a client, while another page's data write token is being revoked by a call from the file server. In this case, part of the execution of the data token revocation call involves a call back to the file server to store the appropriate data page. The response to this call includes the updated file's status, which must be inserted into the cached vnode when the call completes.

In essence, two concurrent store data calls are being made to the file server. One is a normal store data call, while the other is a special call issued only by token revocation code. Both *store* calls send data back to the server and receive as a response the updated file status information. Because the low-level lock is never held over normal client-initiated RPCs, the two responses may be processed in any order. Again, the file serialization counter is used to correctly serialize the responses. As soon as one response is received, the client obtains the low-level lock, and copies the updated status information back into the vnode, but only if the updated status information is labelled with a file serialization counter larger than that already labelling the vnode. After the data are copied, the vnode is labelled with the file serialization counter from the just-merged status. If the counter associated with the vnode is greater than the counter associated with the returned status information, then the returned status information is older and can be simply ignored.

6.4 Other considerations

After any system call completes, calls made via the cache manager will reflect the state of the file at the time the call completed, or some later time. Because old file status information is never written back over a file's vnode, once the up-to-date information is present in the cache manager's vnode, it can not be replaced with older data.

Note that when the cache manager's token revocation procedure calls back to the file server, the cache manager must ensure that some dedicated server threads are available to handle these requests. If only one pool of threads were available for all incoming requests, then it would be possible for all of the server threads to be busy when a token revocation procedure has to call back to the server, resulting in a deadlock.

In general, calls made to the file server from executing client revocation code could be eliminated in place of somewhat complex return parameters from the token revocation procedures. However, this has some aesthetic difficulties, as well as one fundamental difficulty: it is difficult for a server thread to ensure that the last few packets of its reply have been received properly by its caller; this check doesn't quite fit into the RPC paradigm. It turns out to be considerably simpler for a token revocation return call to simply call back to the file server to accomplish such things as returning unused file locks (when revoking lock tokens) or storing back modified data (when revoking write data tokens).

Deadlock avoidance for the remaining subsystems (e.g. the volume server, the volume location server) is quite simple, since in general a client will lock its own resources in some standard order, call a server, which will lock its own resources in its own standard order, and then execute without making any return calls to the client. Thus there are no tricky cycles between the clients and the servers of a particular service.

Summary

To put the DEcorum design in perspective, we observe that the concept of a network file system is not new. Though some obvious applications of such file systems are for LAN's, AFS was specifically designed for networks of thousands of users, and we are still learning what new problems are introduced by the increase of scale. The DEcorum design focuses on three of those problems, selected for their practical and theoretical importance: interoperability; cache consistency; and server availability. It also provides very substantial POSIX compliance and does so in a highly modular fashion. DEcorum borrows heavily from the well-understood design of AFS. However, DEcorum has the potential to grow into much larger environments: users may attempt to create network file systems of national or international size and scope. We may look forward to these challenges with confidence in the robustness and flexibility of our present designs.

Trademarks

AFS is a trademark of Transarc Corporation. UNIX is a registered trademark of AT&T. NFS is a trademark of Sun MicroSystems, Inc.

Acknowledgments

We are grateful to Alfred Spector, Liz Hines, and Julie Kownacki for comments and corrections. We are deeply indebted to the staff of Transarc Corporation for support and encouragement. Ideas from many people are reflected in the ultimate file system design. Some of these people are from the customers and co-developers of AFS, including Carnegie Mellon, MIT, the University of Michigan, and IBM Rochester. Other individuals include (but are not limited to) the following: Bruce Walker, Charlie Sauer, Paul Leach, Mike West, Todd Smith, Don Johnson, Joe Pato, Gary Owens, and Bill Sommerfeld.

Michael L. Kazar is Manager of File Systems Architecture at Transarc Corp. of which he was a founder. This work is an outgrowth of his previous work at Carnegie Mellon University's Information Technology Center, where he was instrumental in the design and implementation of the Andrew File System. He received two S.B. degrees from the Massachusetts Institute of Technology, and received a Ph.D. in Computer Science from Carnegie Mellon University.

Bruce W. Leverett is a Systems Designer at Transarc Corp. Prior to joining Transarc, he worked with Scribe Systems (formerly Unilogic) on electronic publishing products. He received a Ph.D. in Computer Science from Carnegie Mellon in 1981, and an A.B. from Harvard in 1973.

Owen T. Anderson is a Systems Designer at Transarc Corp. Prior to joining Transarc, he was a member of the Andrew File System group at Carnegie Mellon University's Information Technology Center. He had previously worked for ten years at the Lawrence Livermore National Laboratory in Livermore, California. He graduated from the Massachusetts Institute of Technology in 1979 with an S.B. degree in Physics.

Vasilis Apostolides is a Systems Designer at Transarc Corp. He received B.S. and M.S. degrees from Westchester University, and an M.S. degree from the University of Pittsburgh, all in Computer Science. After graduate work, he taught for a year and a half at Westchester, took graduate classes at Temple, and did systems programming for various University organizations before coming to the University of Pittsburgh. For the past four years prior to joining Transarc, he has been working at Carnegie Mellon University on different aspects of the Andrew environment, particularly the Andrew File System.

Beth A. Bottos is a Systems Designer at Transarc Corp. Prior to joining Transarc, she worked as a Research Programmer at Carnegie-Mellon University and as a Member of Technical Staff at A.T.&T. Bell Laboratories. She holds an M.S. degree in Computer Science from CMU, an M.S. degree in Electrical Engineering from Stanford University, and a B.S. degree in Electrical Engineering from Purdue University.

Sailesh Chutani is a Systems Designer at Transarc Corp. He has been involved with the Andrew File System project since June 1988 when he joined Carnegie Mellon University's Information Technology Center. He holds an M.S. in Computer Science from the University of North Carolina at Chapel Hill and a B.S. in Computer Science and Engineering from the Indian Institute of Technology at Kanpur, India.

Craig Everhart is a Senior Systems Designer at Transarc Corp. Prior to joining Transarc, he was a Systems Designer for four years at the Information Technlogy Center at Carnegie Mellon. There he participated in the design and development of the Andrew Message System, a distributed, multi-media electronic mail system that operates as a client both of the Andrew File System and of the Andrew Toolkit. He holds the B.A. in Physics from Wesleyan University as well as the Ph.D. from Carnegie Mellon (Computer Science).

W. Anthony Mason is a Systems Designer at Transarc Corp. Prior to joining Transarc, he worked as a Systems Programmer in the Distributed Systems Group at Stanford University in the Department of Computer Science. He was involved in the development of both the V distributed system and the VMTP transport protocol. He received a B.S. degree in Mathematics from the University of Chicago.

Shu-Tsui Tu is a Systems Designer at Transarc Corp. Prior to joining Transarc, he worked on the AIX Journaled File System and later on BSD44 FS at OSF. Before that, Tu spent five years at Prime Computer, Inc. working on the development of the database management systems, recovery and concurrency control and transactional Unix File Systems. Mr. Tu received his M.S. degree in Computer Engineering from the University of Massachusetts in Amherst.

Edward R. Zayas is one of the original members of the Transarc File Systems Development group. Prior to joining Transarc at its inception, he was a System Designer at the Carnegie Mellon University Information Technology Center for over two years. At the

CMU ITC, he developed the Cellular Andrew concept, whereby administratively autonomous sites running AFS cooperate to provide a single, global file system namespace. He received a Ph.D. in Computer Science from CMU in 1987, an M.S. from CMU in 1983, and a B.A. in Computer Science/Mathematical Statistics from Columbia University in 1979.

USENIX Summer Conference
June 11-15, 1990
Anaheim, California

The COPS Security Checker System

Daniel Farmer – Computer Emergency Response Team, CMU
Eugene H. Spafford – Purdue University

ABSTRACT

In the past several years, there have been a large number of published works that have graphically described a wide variety of security problems particular to UNIX. Without fail, the same problems have been discussed over and over again, describing the problems with SUID (set user ID) programs, improper file permissions, and bad passwords (to name a few). There are two common characteristics to each of these problems: first, they are usually simple to correct, if found; second, they are fairly easy to detect.

Since almost all UNIX systems have fairly equivalent problems, it seems appropriate to create a tool to detect potential security problems as an aid to system administrators. This paper describes one such tool: COPS. COPS (Computerized Oracle and Password System) is a freely-available, reconfigurable set of programs and shell scripts that enable system administrators to check for possible security holes in their UNIX systems.

This paper briefly describes the COPS system. Included are the underlying design goals, the functions provided by the tool, possible extensions, and some experiences gained from its use. We also include information on how to obtain a copy of the initial COPS release.

Introduction

The task of making a computer system secure is a difficult one. To make a system secure means to protect the information from disclosure; protecting it from alteration; preventing others from denying access to the machine, its services, and its data; preventing degradation of services that are present; protecting against unauthorized changes; and protecting against unauthorized access.

To achieve all these security goals in an actual, dynamic environment such as that presented by most UNIX systems can be a major challenge. Practical concerns for flexibility and adaptability render most formal security methods inapplicable, and the variability of system configuration and system administrator training make "cookbook" methods too limited. Many necessary security administration tasks can be enhanced through the use of software and hardware mechanisms put in place to regulate and monitor access by users and user programs. Those same mechanisms and procedures, however, constrain the ability of users to share information and to cooperate on projects. As such, most computer systems have a range of options available to help secure the system. Choosing some options allows enhanced sharing of information and resources, thus leading to a better collaborative environment, where other settings restrict that access and can help make the system more secure.

One of the tasks of a system and security administrator is to choose the settings for a given system so that security is at an appropriate level—a level that does not unduly discourage what sharing is necessary for tasks to be accomplished, but that also gives a reasonable assurance of safety. This often leads to problems when a system has a very wide range of possible settings, and when system administrators lack sufficient training and experience to know what appropriate settings are to be applied.

Ideally, there should be some kind of assistance for system administrators that guides them in the application of security measures appropriate for their environment. Such a system needs to be configurable so it provides the appropriate level of assistance based on the perceived need for security in that environment. That system should be comprehensive enough so that an untrained or inexperienced administrator is able to derive a high degree of confidence that all appropriate features and weaknesses are identified and addressed.

Unfortunately, such a tool may also present a danger to that same system administrator. For instance, there could be a danger if the tool were to fall into the hands of a potential attacker. The tool could be used to analyze the target system or to provide clues for methods of attack. A second potential danger is that the tool can be modified by an unfriendly agent so that the information it reports and the actions that it takes serve not to enhance the security of the system, but to weaken it. A third possibility is that the tool is not comprehensive enough, or that changes in system operation are such that the tool does not expose the security flaws made present by those changes; the security administrator, by relying on the tool, fails to be aware of the new dangers to his or her system.

A good example of all three dangers might be the development and use of a tool that examines passwords to see if they can be easily guessed by an attacker. Such a tool might consist of a fast implementation of the password encryption algorithm used on a particular machine. Provided with this tool would be a dictionary of words that would be compared against user passwords. Passwords that match a word in the dictionary would be flagged as weak passwords.

Such a tool would enable a system administrator to notify users with weak passwords that they should choose a password that is more difficult for an attacker to guess. However, such a tool is a danger to the very same system it is designed to protect should it fall into the hands of an attacker: the tool could be used to very rapidly search through the dictionary in an attempt to find a password that could be compromised.

A second potential danger is that an attacker with sufficient privilege might alter the encryption algorithm or the internal workings of the program such that it would appear to run correctly, but would fail to match certain passwords or certain accounts. This would allow a determined attacker to plant an account with a known simple password that would not be detected by the program. Alternatively, an attacker might modify such a program to send its output to not only the administrator, but to the attacker as well.

The third problem is that the system administrator may grow complacent by running this password tool if it continually reports that there are no weak passwords found. The administrator may not make any effort to enhance the quality or size of the dictionary, or to provide other tracking or audit mechanisms to observe individuals who may be attempting to guess passwords or break into accounts.

For all of these reasons, such a tool might be considered to lessen the overall security of the system rather than to enhance it. That should not prevent us from developing security tools, however. Instead, the challenge is to build tools that enhance security without posing too great a threat when employed by an enemy.

COPS Design and Structure

Design

Although there is no reasonable way that all security problems can be solved on any arbitrary UNIX system, administrators and systems programmers can be assisted by a software security tool. COPS is an attempt to address as many potential security problems as possible in an efficient, portable, and above all, in a reliable and safe way. The main goal of COPS is one of prevention; it tries to anticipate and eliminate security problems by detecting problems and denying enemies an opportunity to compromise security in the first place.

The potential security hazards that COPS checks for were selected from readings of a variety of security papers and books (see the references section at the end of the paper), from interviews with experienced system administrators, and from reports of actual system breakins.

We applied the following important guiding principles to the design and development of COPS:

- COPS should be configurable so that new tools could be added or the existing tools altered to meet the security needs of the installation on which it is run. Since UNIX is so dynamic, it must be possible to incorporate both new tools and methods in COPS as the need for them becomes apparent.
- COPS should contain no tool that attempts to fix any security problems that are discovered. Because COPS makes no modifications to the system, it is not required that it be run with any particular privilege, and many of the tools can be run with privilege less than or equal to that of a regular user. As a result, this lessens the temptation for an intruder to modify the code in an attempt to make surreptitious changes to the system.
- While COPS should notify the administrator that there may be a weakness, it does not describe why this is a problem or how to exploit it. Such descriptions should be found in alternative sources that are not embedded in the program. Thus, a determined attacker might run the program, might be able to read the output, but be unaware of a method to exploit anything that COPS reports it has found.
- COPS should not include any tools whose use by determined attackers, either standalone or as part of the COPS system, would give them a significant advantage at finding a way to break into the system beyond what they might already have in their possession. Thus, a password checking tool, as was previously described, is included, but the algorithm utilized is simply what is already present in the

system library of the target system.

- COPS should consist of tools and methods that are simple to read, understand, and to utilize. By creating the tools in such a manner, any system administrator can read and understand the system. Not only does this make it easier to modify the system for particular site needs, but it allows reexamination of the code at any time to ensure the absence of any Trojan horse or logic bomb.
- The system should not require a security clearance, export license, execution of a software license, or other restriction on use. For maximum effectiveness, the system should be widely circulated and freely available. At the same time, users making site-specific enhancements or including proprietary code for local software should not be forced to disclose their changes. Thus, COPS is built from new code without licensing restrictions or onerous "copyleft," and bears no restriction on distribution or use beyond preventing it from being sold as a commercial product.
- COPS should be be written to be portable to as wide a variety of UNIX systems as possible, with little or no modification.

In order to maximize portability, flexibility, and readability, the programs that make up COPS are written as simple Bourne shell scripts using common UNIX commands (`awk`, `sed`, etc.), and when necessary, small, heavily-commented C programs.

Structure

COPS is structured as a dozen sub-programs invoked by a shell script. That top-level script collects any output from the subprograms and either mails the information to the local administrator or else logs it to a file. A separate program that checks for SUID files is usually run independently because of the amount of time required for it to search through the filesystems. All of the tools except the SUID checker are not meant to be run as user root or any other privileged account.

Please note that the descriptions of the tools provided here do not contain any detailed explanation of why the tools check what they do. In most cases, the reason is obvious to anyone familiar with UNIX. In those cases where it is not obvious, the bibliographic material at the end of this paper may provide adequate explanations. We apologize if the reasons are not explained to your satisfaction, but we do not wish to provide detailed information for potential system crackers who might have our system.

These are the individual the programs that comprise COPS:

dir.check, file.chk These two programs check a list of directories and files (respectively) listed in a configuration file to ensure that they are not world-writable. Typically, the files checked would include `/etc/passwd`, `/.profile`, `/etc/rc`, and other key files; directories might include `/`, `/bin`, `/usr/adm`, `/etc` and other critical directories.

pass.chk This program searches for and detects poor password choices. This includes passwords identical to the login or user name, some common words, etc. This uses the standard library crypt routine, although the system administrator can link in a faster version, if one is available locally.

group.chk, passwd.chk These two tools check the password file (`/etc/passwd` and `yppasswd` output, if applicable) and group file (`/etc/group` and `ypgroup` output, if applicable) for a variety of problems including blank lines, null passwords, non-standard field entries, non-root accounts with uid=0, and other common problems.

cron.chk, rc.chk These programs ensure that none of the files or programs that are run by `cron` or that are referenced in the `/etc/rc*` files are world-writable. This protects against an attacker who might try to modify any programs or data files that are run with root privileges at the time of system startup. These routines extract file names from the scripts and apply a check similar to that in file.chk.

dev.chk checks `/dev/kmem`, `/dev/mem`, and file systems listed in `/etc/fstab` for world read/writability. This prevents would-be attackers from getting around file permissions and reading/writing directly from the device or system memory.

home.chk, user.chk These programs check each user's home directory and initialization files (`.login`, `.cshrc`, `.profile`, etc.) for world writability.

root.chk This checks root startup files (e.g., `/.login` and `/.profile`) for incorrect `umask` settings and search paths containing the current directory. This also examines `/etc/hosts.equiv` for too much accessibility, and a few miscellaneous other tests that do not fit anywhere else.

suid.chk This program searches for changes in SUID file status on a system. It needs to be run as root for best results. This is because it needs to find all SUID files on the machine, including those that are in directories that are not generally accessible. It uses its previous run as a reference for detecting new, deleted, or changed

SUID files.

kuang The U-Kuang expert system, originally written by Robert W. Baldwin of MIT. This program checks to see if a given user (by default, root) is compromisable, given that certain rules are true.

It is important to note once again that COPS does not attempt to correct any potential security hazards that it finds, but rather reports them to the administrator. The rationale for this is that is that even though two sites may have the same underlying hardware and version of UNIX, it does not mean that the administrators of those sites will have the same security concerns. What is standard policy at one site may be an unthinkable risk at another, depending upon the nature of the work being done, the information stored on the computer, and the users of the system. It also means that the system does not need to be run as a privileged user, and it is less likely to be booby-trapped by a vandal.

Usage

Installing and running COPS on a system usually takes less than an hour, depending on the administrator's experience, the speed of the machine, and what options are used. After the initial installation, COPS usually takes a few minutes to run. This time is heavily dependent on processor speed, how many password checking options are used, and how many accounts are on the system.

The best way to use COPS is to run it on a regular basis, via `at` or `cron`. Even though it may not find any problems immediately, the types of problems and holes it can detect could occur at any later time.

Though COPS is publically accessible, it is a good idea to prevent others from accessing the programs in the toolkit, as well as seeing any security reports generated when COPS has been run. Even if you do not think of them as important, someone else might use the information against your system. Because COPS is configurable, an intruder could easily change the paths and files that COPS checks, thus making any security checks misleading or worthless. You must also assume intruders will have access to the same toolkit, and hence access to the same information on your security problems. Any security decisions you make based on output from COPS should reflect this. When dealing with the security of your system, caution is never wasted.

Experience and Evaluation

This security system is not glamorous—it cannot draw any pictures, it consists of a handful of simple shell scripts, it does not produce lengthy, detailed reports, and it is likely to be of little interest to experienced security administrators who have already created their own security toolkits. On the other hand, it has proven to be quite effective at pointing out potential security problems on a wide variety of systems, and should prove to be fairly valuable to the majority of system administrators who don't have the time to create their own system. Some administrators of major sites have informed us that they are incorporating their old security checks into COPS to form a unified security system.

COPS has been in formal release for only a few months. We have received some feedback from sites using the system, including academic, government and commercial sites. All of the comments about the ease of use, the readability of the code, and the range of things checked by the system have been quite positive. We have also, unfortunately, had a few reports that COPS may have been used to aid in vandalizing systems by exposing ways to break in. In one case, the vandal used COPS to find a user directory with protection modes 777. In the other case, the vandal used COPS to find a writable system directory. Note, however, that in both of these cases, the same vulnerability could have easily been found without COPS.

It is interesting to note that in the sites we have tested, and from what limited feedback we received from people who have utilized it, over half the systems had security problems that could compromise root. Whether that can be generalized to a larger population of UNIX systems is unknown; part of our ongoing research is to determine how vulnerable a typical site may be. Even machines that have come straight from the vendor are not immune from procedural security problems. Critical files and directories are often left world-writable, and configuration files are shipped so that any other machine hooked up to the same network can compromise the system. It underscores this sad state of affairs when one vendor's operational manual harshly criticizes the practice of placing the current directory in the search path, and then in the next sentence states "Unfortunately, this safe path isn't the default."[1]

We plan on collecting further reports from users about their experiences with COPS. We would encourage readers of this paper who may use COPS to inform us of the performance of the system, the nature of problems indicated by the system, and of any suggestions for enhancing the system.

Future Work

From the beginning of this project, there have been two key ideas that have helped focus our attention and refine our design. First, there is simply no reasonable way for us to write a security package

[1] We will not embarrass that one vendor by citing the source of the quote. At least they noted the fact that such a path is a hazard; many vendors do not even provide that much warning.

that will perform every task that we felt was necessary to create a truly satisfactory security package. Second, if we waited, no one else was going to write something like COPS for us. Thus, we forged ahead with the design and construction of a solid, basic security package that could be easily expanded. We have tried to stress certain important principles in the design of the system, so that the expansion and evolution of COPS will continue to provide a workable tool.

COPS was written to be rewritten. Every part of the package is designed to be replaced easily; every program has room for improvement. The framework has room for many more checks. It seems remarkable that a system as simple as COPS finds so many flaws in a typical UNIX installation! Nonetheless, we have thought of a number of possible extensions and additions to the COPS system; these are described in the following sections.

Detecting known bugs

This is a very difficult area to consider, because there are an alarming number of sites (especially commercial ones) without the source code that is necessary to fix bugs. Providing checks for known bugs might make COPS more dangerous, thus violating our explicit design goals. At the same time, checking for known bugs could be very useful to administrators at sites with access to source code.

If we keep in mind that COPS is intended as a system for regular use by an administrator, we conclude that checking for known bugs is not appropriate in COPS, because such checks are ordinarily done once and not repeated. Thus, a separate system for checking known bugs would be appropriate—a a system that might be distributed in a more controlled manner. We are currently considering different methods of distributing such a system.

Checksums and Signatures

Checksums and cryptographically-generated signatures could be an excellent method of ensuring that important files and programs have not been compromised. COPS could be enhanced to regenerate these checksums and compare them against existing references. To build this into COPS will require some method of protecting both the checksum generator and the stored checksums, however. It also poses the problem that system administrators might rely on this mechanism too much and fail to do other forms of checking, especially in situations where new software is added to the system.

Detecting changes in important files

There are some files that should change infrequently or not at all. The files involved vary from site to site. COPS could easily be modified to check these files and notify the system administrator of changes in contents or modification times. Again, this presents problems with the protection of the reference standard, and with possible complacency.

NFS and Yellow Pages

Many new vulnerabilities exist in networked environments because of these services. Their recent development and deployment mean that there are likely to be more vulnerabilities and bugs present than would be found in more mature code. As weaknesses are reported, corresponding checks should be added to the COPS code.

Include UUCP security checks

Because UUCP is very widely used, it is important to increase the number and sophistication of the checks performed on all the different varieties of UUCP. This includes checking the files that limit what programs can be remotely executed, the `USERFILE` and `L.sys` files, and the protections on directories.

Configuration files

There are many problems that result from improper configuration files. These occur not only from having the files open to modification, but because of unexpected or misunderstood interactions of options. Having rule-based programs, similar to kuang, that analyze these configuration files would be an ideal way to extend COPS.

Checking OS-specific problems

There are a wide variety of problems that apply only to certain flavors of UNIX. This includes not only the placement of key files, but also syntactical and logical differences in the way those systems operate. Examples include such things as shadow password files, different system logging procedures, shared memory, and network connectivity. Ideally, the same set of tools would be used on every system, and a configuration file or script would resolve any differences.

Conclusions

Over the last 18 months since the Internet worm, perhaps the most strongly voiced opinion from the Internet community has been "security through secrecy does not work." Nonetheless, there is still an appalling lack of communication about security. System breakers and troublemakers, on the other hand, appear to encounter little difficulty finding the time, energy, and resources necessary to break into systems and cause trouble. It is not that they are particularly bright; indeed, examining the log of a typical breakin shows that they follow the same methods that are publicized in the latest computer security mailing lists, in widely publicized works on UNIX security, and on various clandestine bulletin boards. The difference between them and the system administrators on the Internet seems to be communication. It is clear that the underground community has a well-established pipeline of information that is relatively easy for them to tap. Many system administrators, however, have no access to an equivalent source of information, and are thrust into their positions with little or no security experience. COPS should be particularily helpful in these cases.

None of programs in COPS cover all of the possible areas where a system can be harmed or compromised. It can, however, aid administrators in locating some of the potential trouble spots. COPS is not meant to be a panacea for all UNIX security woes, but an administrator who examines the system and its documentation might reduce the danger to their UNIX system. That is all that can ever be expected of any security tool in a real, operational environment.

Future work on COPS will be done at the CERT, and work on related tools and approaches will be done at Purdue. People are encouraged to get a copy of COPS and provide us with feedback. We expect that as time goes on, and as the awareness of security grows, COPS and systems like it will be evolved through community effort. Increased communication and awareness of the problems should not be limited to just the crackers.

Acknowledgments

Thanks go to Robert Baldwin for allowing us to include his marvelous U-Kuang system; to Donald Knuth for inspirational work on how not only to write but to create a software system; to Jeff Smith, Dan Trinkle, and Steve Romig for making available their systems and expertise during the development of COPS; and finally, our beta testers, without which COPS might never have been.

Appendix—Getting COPS

COPS has been run successfully on a large number of computers, including UNIX boxes from Sun, DEC, HP, IBM, AT&T, Sequent, Gould, NeXT, and MIPS.

A copy of COPS was posted to the comp.sources.UNIX newsgroup and thus is available in the UUCP archives for that group, as well as via anonymous ftp from a variety of sites (uunet.uu.net and j.cc.purdue.edu, for example.) We regretfully cannot mail copies of COPS to sites, as we do not have the time or resources to handle such requests.

References

Aho, Aho, Alfred V., Brian W. Kernighan, and Peter J. Weinberger, *The AWK Programming Language,* Addison-Wesley Publishing Company, 1988.

Authors, Authors, Various, *UNIX Security Mailing List/Security Digest,* December 1984-present.

Baldwin, Baldwin, Robert W., *Rule Based Analysis of Computer Security,* Massachusetts Institute of Technology, June 1987.

Grampp, Grampp, F. T. and R. H. Morris, "UNIX Operating System Security," *AT&T Bell Laboratories Technical Journal,* October 1984.

Kaplilow, Kaplilow, Sharon A. and Mikhail Cherepov, "Quest—A Security Auditing Tool," *AT&T Bell Laboratories Technical Journal,* AT&T Bell Laboratories Technical Journal, May/June 1988.

Smith, Smith, Kirk, "Tales of the Damned," *UNIX Review,* February 1988.

Spafford, Spafford, Eugene, Kathleen Heaphy and David Ferbrache, *Computer Viruses: Dealing with Electronic Vandalism and Programmed Threats,* ADPASO, 1989.

Spence, Spence, Bruce, "spy: A UNIX File System Security Monitor," *Proceedings of the Large Installation Systems Administration III Workshop,* September, 1988. ,IP Thompson, 5 Thompson, Ken, "Reflections on Trusting Trust," **27**(8), *Communications of the ACM,* August 1984.

Wood, Wood, Patrick and Stephen Kochran, *UNIX System Security,* Hayden Books, 1986.

Wood, Wood, Patrick, "A Loss of Innocence," *UNIX Review,* February 1988.

Dan Farmer is a member of the CERT (Computer Emergency Response Team) at the Software Engineering institute at Carnegie Mellon University. He is currently designing a tool that will detect known bugs on a variety of UNIX systems, as well as continuing program development and design on the COPS system. His address is: Software Engineering Institute, Carnegie Mellon University; Pittsburgh, PA 15213-3890. Reach him electronically at df@sei.cmu.edu.

Gene Spafford is an assistant professor at Purdue University in the Department of Computer Sciences. He is actively involved with software engineering research, including testing and debugging technology. He is also actively involved in issues of computer security, computer crime, and professional ethics. Spaf is coauthor of a recent book on computer viruses, and is well-known for his analysis of the Morris Internet Worm. Besides being a part-time netgod, Gene is involved with ACM, IEEE-CS, the Computer Security Institute, the Research Center on Computers and Society, and (of course) Usenix. His address is: Purdue University; West Lafayette, IN 47907-2004. Reach him electronically at spaf@cs.purdue.edu.

USENIX Summer Conference
June 11-15, 1990
Anaheim, California

The Evolution of *turnin*: A Classroom Oriented File Exchange Service

William Cattey – MIT Project Athena

ABSTRACT

From the beginning, MIT Professors wanted to use the Project Athena campus wide computer network to collect assignments. The *turnin* program started off as a shell script that sent files to a central timesharing host for perusal by a grader. When timesharing hosts were replaced by a network of workstations, *turnin* became a network service layered on NFS. At that time, programs were added to help graders sort through the files. Student retrieval of prepared handouts and the exchange of papers in real-time were added. Today, *turnin* is a stand alone network service with cooperating servers and a replicated database. It has been integrated with a WYSIWYG editor for composition of complex documents, and their annotation by peers and teachers. The teacher side of the interface is evolving into a point and click gradebook interface. This paper describes the evolution of *turnin* from its original shell script form to its current integrated editing/formatting/annotating system.

Introduction

Project Athena exists to deploy and maintain a campus-wide network of workstations and services (such as remote file access, printing, and electronic mail) to MIT for educational use. In addition to deploying workstations and linking them together with the network, Athena develops software of general use to the MIT community, for example new system services (such as the *Zephyr* notification service [DellaFera1988a]), and educational software programs such as TODÓR [Murman1988a].

The *turnin* system is both a file exchange system service and educational software. It was developed in response to the request of MIT professors who wanted to use Athena to manage electronically submitted assignments. The first version supported student submission of assignments, and teacher return of annotated assignments or new files.

Subsequent versions offered more capabilities: real-time exchange of files in class, distribution of handouts, organization of papers to grade for the teacher, and integration of file exchange with document preparation.

This paper chronicles *turnin* from its beginning as a ‘‘silly little rsh script’’ to its present integrated application.

Version 1: "The rsh Hack"

The first version of the *turnin* service had the least functionality, the worst user interface, and the most difficult set up process. It was first used in the early

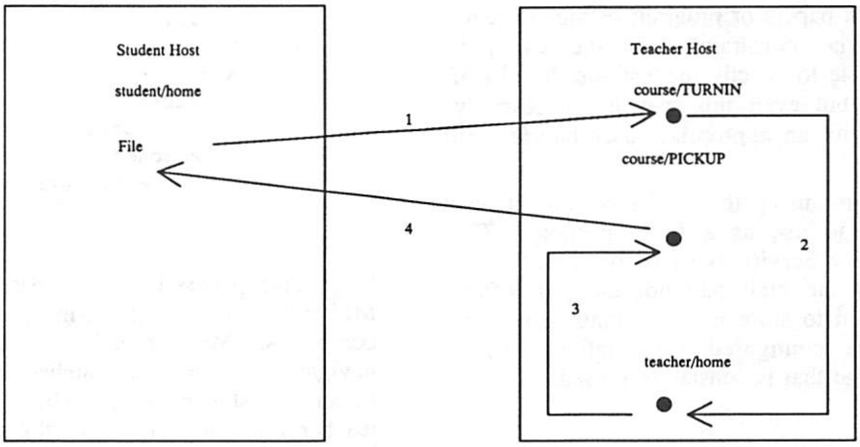


Figure 1: The Paper Path

days of Athena before workstations were installed. At that time Athena consisted of 63 networked timesharing hosts.

The path followed by student papers is shown in Figure 1. Files exchanged through *turnin* would start in the student's home directory on their assigned timesharing host. The *turnin* program would send the file over the net to a sub-directory associated with the course on the teacher's timesharing host. The teacher would find the file, probably move it to his or her home directory, manipulate it, (perhaps compile it, perhaps edit it), and then move it to the course's pickup subdirectory, where the student could find it when he or she ran the *pickup* program.

Why not electronic mail?

Other communication services developed at Athena such as *discuss* [Raeburn1989a], and *Zephyr* [DellaFera1988a] were not implemented as electronic mail because they required services that electronic mail did not provide. (Instantaneous transmission in the case of *Zephyr*, and central sequenced storage of messages in the case of *discuss*.) With *turnin*, the primary issue was user interface.

Professors wanted a user interface that more closely resembled the traditional "hand it in, mark it up, give it back" classroom process than existing exchange systems such as electronic mail and network filesystems did.

The service could be built on top of a network file system, the mailer, or any other available communication system as long as professors and students felt comfortable exchanging papers with it. For example, they didn't want to deal with mail headers in papers. In the simplest application, students would compose documents. These documents would be annotated by the teachers. The annotated documents would, probably, be re-used by the students for later drafts.

Some professors wanted to receive executable files to run rather than papers or program listings to read. This imposed the constraint that the transport mechanism be able to exactly reconstitute the bits of the submission, but even this was not necessarily difficult to do with an appropriate user interface to the mailer.

We decided against using the mailer because it was not well suited to use as a file repository. The Athena Post Office Service is based on the assumption that neither the mail hub nor the post office machines are used to store mail for long periods of time. They are configured for relatively small amounts of storage that is constantly reused.

The Student User Interface

Students were expected to use the UNIX timesharing host to do their work and exchange it with professors using these two commands:

```
turnin problem_set file [file]
```

and

```
pickup [problem_set]
```

The `problem_set` argument was a string to name the problem set. Often this would be specified by the teacher. The file arguments could also be directories. If *pickup* were called with no argument or if the named problem set was not found, a list of existing problem sets to pickup was returned.

The Teacher Non-interface

The only professors who used *turnin* were UNIX cognoscenti or had teaching assistants who would do all the computer work for them. One course only used *turnin* as a way to collect assignments. A TA would send the assignments to the line printer and take the printouts to the grading meeting.

To annotate files the teacher was expected to know the *turnin* file hierarchy and to use UNIX commands to obtain the file, edit it, and save the changed file in a similarly structured pickup hierarchy.

Here is an example of what the file hierarchy might look like for a course, say, intro with two students jack and jill:

```
intro/
    TURNIN/
        jack/
            first/
                README
                foo.c
            second/
                Makefile
                foo1.c
                foo2.c
        jill/
            ...
    PICKUP/
        jack/
            first/
                foo.errs
            second/
                a.out.errs
        jill/
            ...
```

Organizing papers this way posed a problem. Most MIT professors are expert in their own fields, not in computers. Many found the hierarchy difficult to navigate and not well matched to their methods of organizing student work. This tended to make the teacher's job harder, while the goal of *turnin* had been to make it easier.

The Transport Mechanism

The first version of *turnin* relied on a primitive mechanism for transfering data. Users of Berkeley UNIX may be familiar with the following command line to duplicate a hierarchy on a remote host:

```
tar cf - | rsh remote.host \
   ''(cd destination/directory; tar xpBf -)''
```

On the timesharing host that was to receive papers, a special account called "grader" was created. Instead of /bin/csh like everyone else, grader's login shell was *grader_tar*. This program relied on receiving as arguments:

- a flag to determine if this was a turnin or a pickup
- the student's username
- the hostname of the machine the student was on
- a name for the problem set
- the absolute pathname of the student's working directory
- the name of the file or directory being transferred.

It used this information to locate the files to transfer, and to set the student's host as the remote.host to rsh to. The *turnin* program would `rsh -l grader` passing it these arguments, and the *grader_tar* program would rsh back to the host that initiated the *turnin* to perform the transmission!

Security

Care was taken to prevent unauthorized access to turned-in files. The course *turnin* directory was accessible only to members of a file protection group which was specially made for each course, and to the magic grader account.

The grader.tar program relied on rsh for security. The *turnin* program would modify a .rhosts file in the student's home directory to allow the grader.tar's rsh to succeed. There was no global trusting among the timesharing hosts.

Probably the best enforcement of security came from the obscurity of the program. It was hard for the installers and developers to retain a clear idea of how it worked. This security through obscurity proved adequate for our prototype, but it was unacceptable for a *turnin* service in wide use.

Problems

The *turnin* prototype was an interesting experiment in using the Athena timesharing hosts to exchange papers electronically, but as a service it left much to be desired. It had problems in setup, maintainability, and usability.

The maintenance headaches came from teaching new people how to set up *turnin*, and in controlling the amount of disk taken up by the course. Setup required establishment of the grader account on the timesharing host, and installation of the user programs in course program libraries. The location of the course *turnin* directory had to be established and placed in a file along with the *turnin* program in the course program libraries. Athena User Accounts had to create a group for the graders, and keep it up to date. Student user id's had to be known to the course timesharing host, even if the students were not allowed to log on to it. Disk consumption had to be monitored by a person periodically running *du* over the course directory.

Usability was the biggest problem. A grader interface for those who were not UNIX literate was needed. It had to help rather than hinder the organizing of student files. The time delay in depositing files needed to be reduced. Other activities closely related to collecting and returning papers could also be supported.

The real reason why *turnin* version two came about, however, was the demise of the timesharing hosts, and the elimination of rsh as a transport mechanism.

Version 2: Expanded Functionality

In spite of the problems with the user interface, *turnin* acquired a small but loyal following of satisfied customers. Athena decided to try and keep these customers when the timesharing model of computing was replaced with the distributed services model. Our challenge as *turnin* developers was to find a new file exchange mechanism not based on rsh that would work in the environment of non-secure workstations contacting secure service hosts.

The person assigned to do the new version of *turnin* was concurrently working with the MIT writing faculty on another project. He saw an opportunity to expand turnin; to market it to new users while at the same time completely re-implementing the old service. This follows the great MIT tradition, "We can't solve that problem, but we can solve another much harder problem."

The MIT writing faculty had formed a group, the Committee on Writing Instruction and Computers (known as CWIC, pronounced "quick"). This group met weekly to discuss ways that computers, specifically those offered by Athena, could be applied to improve writing instruction at MIT. This group came up with four basic activities of writing instruction they wanted the computer to support. In a writing class, students and teachers:

- create texts
- exchange texts
- display text
- critique, annotate, and discuss texts

A specification, *Electronic Classroom: Specification for a User Interface* [Barrett1987a], was written based on the group's work. The second and third versions of *turnin* were based on this spec. Starting from the principle of computer enhancement of common activities in class, six components were called for:

1. A **Classroom Put and Get Program** to pass files between students during class.
2. A **Grade Sheet** interface for assignments to provide a simple method for an instructor to organize and correct student assignments.
3. A **Syllabus** interface for handouts to organize these files in an easy to use format and to permit instructors to add or modify curricular files for in-class use.
4. A **Turnin** interface for turning in and picking up assignments and handouts. This interface simplifies current protocols for submitting assignments and picking up annotated papers or other class handouts.
5. An **Electronic Textbook** facility that permits the storage of a set of files representing class notes, instructions and other reference material.

6 A **Presentation Facility** to format files for display on a screen projection device, (i.e. Show the file on the workstation screen in a big font so it will be legible when displayed in class with a screen projection system.)

Component number four represented a re-affirmation that the old *turnin* would be useful. The new version of *turnin* would implement as services the new components one, two, and three as well as re-implementing number four. The ''Grade Sheet interface'' of component two specified how organizing student papers would be improved for teachers not literate in UNIX.

The files managed by the new version of *turnin* were organized into three classes:

- exchangeables -- for component one, the in-class put and get facility
- gradeables -- for components one and four
- handouts -- for component three

The file exchange system was designed to support these and future classes of files so that user interfaces could be tailored to the kind of file being exchanged. The whole system of six components, and the procedures and protocols for applying them in the course of writing instruction were collectively called EOS the Educational On Line System.

Turnin on top of NFS

The spec was completed on 3 June 1987 and handed off to the development team. The team consisted of Maria Camblor, Bruce Lewis, and Rob Shaw with me as project manager. The timesharing model of computation at Athena was to be replaced by the distributed services model of computation on 1 September 1987. What sort of software could we offer as a robust implementation of the spec in three months, given that none of the team members had ever written a network service before?

It was initially unclear what sort of new transport mechanism was to be used in Athena's new distributed services environment. We decided not to use the mail system for reasons described above. Athena had recently deployed its *discuss* service [Raeburn1989a]. We opted not to use the discuss protocol because generating lists of student papers would take a long time, all the papers would be kept in one large file, and utilities to allow old style UNIX command oriented manipulation would be hard to write.

We decided to access the server through a client library (which we named FX.) This would allow the same application programmers interface regardless of what transport mechanism we used. We expected to throw our first server away. We believed that the best way to offer the file exchange service was via a remote procedure call, much like the successful X server. We studied the Sun Remote Procedure Call system and felt it was the right implementation scheme, but were concerned that it would take too long to write and test such a server. As described below, we opted for a much simpler implementation strategy.

We agreed on the FX library calls by studying the spec and by designing the user programs. While Maria Camblor and I set about designing the server and implementing the client library, Rob Shaw and Bruce Lewis wrote the student programs and the teacher programs respectively. We documented our design for use by later developers [Camblor1988a].

The Clients

The student programs ended up resembling the original *turnin* programs. There were three major differences: Assignments were changed to be numbers rather than names. Teachers asked to organize papers by class week number. The course was specifiable by a command line argument and an environment variable. New commands were added for the two new classes of files. The student commands were:

put - store a file in the in-class bin of files to exchange
get - fetch a file from the in-class bin of files to exchange
take - fetch a teacher created handout file
turnin - deliver assignment file
pickup - retrieve corrected assignment files

The student executed these programs from the shell when it was time to fetch or store a file.

The teacher program was started once and had its own command parser. It enabled the teacher to create handouts, administer the class list, and to read, annotate, and return files.

There was a list of commands for each of the three functions shown in the table below.

The teacher would type the command and arguments. At any time the teacher could type "?" and get a list of the commands and a reminder of what the arguments were, or use the info command for more detailed information.

The trickiest part of using the grader program was in specifying files. If no files were specified, the program would assume that all were to be manipulated. To restrict operation the teacher would give a file specification with four parts separated by commas as the argument:

1. assignment number (abbreviated as)
2. author user name (au)
3. version number (vs)
4. file name (fi)

So commands like list taking a file name would have a reminder like this:

```
list [as,au,vs,fi]
```

An empty field matched all, so

```
list 1,wdc,,
```

would list all files turned in by user "wdc" for assignment 1.

The administrative commands were installed to enable teachers to establish a class list. Only students in the class list were allowed to turn in or exchange files. The faculty found it inconvenient to maintain a class list on the computer, and the administrative commands were dropped from the grader program after a short amount of use. The actual workings of access control was de-coupled from the class list as described below.

The important functions were display, annotate, and return. The display command was used for showing files in class. It would fetch the specified file from the server and use a settable display program to view it. (e.g. *more*, *vi* or *emacs*) In practice a special *emacs* with a large font was used as the display program. The annotate and return functions were used to bring the turned-in file into an editor such as *emacs* and to return the modified file to the student. These commands were smart enough to be able to fetch and store multiple files.

The Server

We had insufficient time and experience to write a bona fide server. Instead, the client library attached an NFS filesystem, and implemented all the client calls as file operations. These NFS filesystems would be mounted when the application programs used the `fx_open` call to begin, and unmounted when the `fx_close` call was made, or when the program exited. This operation is consistent with operation of a network service.

We planned to implement access control lists, but abandoned them too due to lack of time and experience. Jon Rochlis looked at the old *turnin* hierarchy and suggested a way to use the standard UNIX file protection modes for access control. The method relied on the newly developed group access authentication that had been added as an Athena local change to NFS.

The NFS filesystems were organized as follows:

- Each course had its own NFS directory which was attachable by name.
- There was a subdirectory for each type of file:

Function	Command(s)	Purpose
grade	list, l	list files turned in
	whois, who	find a student's real name
	display, show	display a file
	annotate, ann	annotate a file
	return, ret, r	return annotated file to student
	editor	change or display current editor
	purge, del, rm	remove turned-in file from bins
	man, info	display information on a command
hand	list, l	list handouts
	whatis, wha	show note for a handout
	put, p	copy a file to a handout
	note, n	add a note to a handout
	take, get, t	copy a handout to a file
	purge, del, rm	remove handouts
admin		add
	del	delete a name
	list, l	list all names in course

turnin, pickup handout, and exchange.

- Each of these directories was group owned by a file protection group created specially for the course.
- All of the graders were in this group, and no student taking the course was.
- The exchange directory was world readable and writable
- The handouts directory was grader writable and world readable
- The *turnin* and *pickup* directories were not world readable, but were world searchable and world writable.
- The first time a student ran *turnin*, a directory owned by him or her, inheriting the group ownership, but inaccessible to the rest of the world, would be created in the *turnin* and *pickup* directories
- All turned in files and files to be picked up would live in the student owned subdirectories.
- We used the 4.3bsd ''sticky bit hack'' to restrict file deletion to directory owners, even when the directory is labeled writable. Students could add themselves to the course but could not delete themselves or anyone else.

This had the effect that students could not find out who else's files were on the server, they could only write files into the *turnin* directory, and would have trouble finding them to modify. They certainly could not read or modify other students files. The graders had free access to all the files. By attaching the course directory by hand, it was possible to create bogus *turnin* directories potentially locking out students. But the perpetrator would own the directories and could be traced.

The file hierarchy looked like this (for a single student, wdc):

```
-r--r--r--  1 jfc  coop   EVERYONE
-rw-r--r--  1 jfc  coop   List
drwxrwxrwt  2 jfc  coop   exchange
drwxrwxr-t  2 jfc  coop   handout
drwxrwx-wt  3 jfc  coop   pickup
drwxrwx-wt  3 jfc  coop   turnin

exchange:

handout:
  -rw-rw-r--  1 wdc  coop    1,wdc,0,avl.h

pickup:
  drwxrwx---  2 wdc  coop   wdc

pickup/wdc:
  -rw-rw-rw-  1 wdc  coop   1,wdc,0,bond.fnd

turnin:
  drwxrwx---  2 wdc  coop   wdc

turnin/wdc:
  -rw-rw----  1 wdc  coop   1,wdc,0,bond.fnd
```

The class list (later abandoned) was kept in the file List. The existence of a file named EVERYONE signified that access was unrestricted. The owner of EVERYONE had to match the owner of the directory it was found in (to prevent just anyone from setting EVERYONE).

Using the clever NFS access modes, we wrote the server and FX client library in two weeks.

We met the user level spec with command line oriented programs: the familiar *turnin* and *pickup*. We added put and get for in-class file exchange, and *take* for teacher created handouts. Our crowning achievement was *grade*, a command oriented subsystem for finding new papers bringing them into an editor, and then returning modified papers.

Problems

This version successfully addressed the problem of usability. The teacher input from CWIC enabled us to provide a better user interface, and enhanced functionality. Retaining the old hierarchy allowed teachers who were comfortable with UNIX to grade as they had in the past, by attaching the NFS course directory manually. The major usability problem remaining was the long time it took to generate lists of files. Since the files were spread across several directories, the FX library did the equivalent of a *find* to locate all the new files. The *turnin* suite of programs were, for the most part, simple, but could have been better integrated into the flow of the classroom. Typing commands could perhaps have been replaced by clicking the mouse.

The problems of setup and maintainability persisted. Controlling disk use became much more of a problem. The new problem of reliability arose: There was no graceful degradation of service in the face of NFS server failure. Disk usage had to be either monitored carefully, or had to have a lot of slack. If one student turned in enough to consume all the disk space, all courses using that NFS partition for *turnin* would be denied service.

Since the *turnin* file repository was a single directory hierarchy manipulated directly by the FX library, there was no mechanism for having secondary storage places if the course directory could not be written into. Even if secondary storage was available, no thought was given to merging primary and secondary storage places, nor to being able to tell when all storage places were accessible. If the one NFS directory was full, or if the particular server was down, that entire course was denied *turnin* service.

The *turnin* service put a strain on the Athena operations staff. The staff was only funded 9AM to 5 PM five days a week. Students would turn papers in 24 hours a day, seven days a week. If the NFS server went down, no paper could be turned in. In order for all courses to perceive *turnin* service to be working, *all* NFS servers holding *turnin* directories had to be working.

The reliability of the NFS based *turnin* system became difficult to maintain near the end of every term when the entire Athena system received its heaviest load. The *turnin* servers became heavily used with students turning in final papers, filling up the course directories when the operations staff is spread thin keeping the whole system up.

Our NFS servers used the Berkeley 4.3bsd quota system used to monitor disk consumption. This implementation of quota clashed with the mechanisms *turnin* used for access control. Since quota was by userid and since access was controlled by having students own their turned in files, quota would have to be set for each individual student. Professors found it onerous to use the grader tools to keep an on-line class list. Having to continuously advise Athena User Accounts of changes in the class was out of the question.

Creating a default quota for all students on a particular server would not have helped much. It would have been difficult to come up with a default number of disk blocks to allocate because some students were in more than one course, and some courses required bigger files than others. Although it would have solved the one failure mode mentioned above of one student denying *turnin* service, other modes of failure occurred more often. We never observed one student turning in a large block of files, but we often observed professors saving all student papers over a term and running the disk out of space.

Until some mechanism of group quota or access control by list rather than owner were developed, quota was disabled for course directories that used *turnin*. Someone on the Athena staff was assigned to watch over the disk usage.

We worked around disk space problems by spreading out course directories among several NFS servers, dedicating large partitions to the non-quota directories, and having one person spend a lot of time watching the disk usage. We tried to limit course directories to 50 meg in a term, and to keep in contact with professors so that they could delete files before space became a problem.

Version 3: The Network Service

Whereas no more than 3 courses used the first version of *turnin*, a dozen used the second version and more were expected in the future. The service was proving useful and was expected to become popular. This meant that something had to be done to solve the operational problems. This stand-alone version was written by Rob French. Bruce Lewis, Clifford Tse, and I contributed.

The new version had to retain or improve upon the previous version's functionality, security, and speed. For example, several courses were exchanging files in class in real time, and collecting handouts at the beginning of class. This real-time performance had to be retained.

The new version had to allow for easier server management. It had to allow:

- simpler setup.
- automated monitoring, and control of disk space usage through some quota mechanism.
- graceful degradation rather than total denial of service in the face of server failures.

We proposed to write a new back end for the FX client library that would meet these requirements.

- It was a true client/server model of service.
- It was layered on top of the Sun remote procedure call protocol.
- It contained its own access control list system.
- Files were owned by the server daemon userid.

The Server

The Sun Remote Procedure Call protocol was a commonly available mechanism for implementing a network service. Other protocols such as the Apollo Remote Procedure Call protocol could have been used. We chose Sun RPC because it was available, Rob had experience with it, and it seemed that its use was becoming more widespread.

The access control lists are maintained in a database under the control of the server. Previously, access control relied on the Athena method of creating credentials files which were updated nightly on all NFS servers. Intervention of Athena User Accounts and a significant time delay were required to offer *turnin* service to new courses, or to modify the list of qualified graders.

With the *turnin* server taking direct responsibility for access control, changes are made through simple applications, and take effect almost instantaneously. The head TA of a course can now add new graders. He or she needs no other special privileges or training. A new course can be created and used right away.

Having the server daemon own all the files is not a very good solution. If each course is arranged as a separate partition or volume, a quota for the daemon userid can be established. It would be a simple matter to make the server run setuid root and to be able to set the file ownership to a particular userid for quota purposes. Perhaps the best solution is to add quota management to the access control lists so that the quota establishment, too, can be an instantaneous process divorced from Athena User Accounts.

We made the following additional changes:

- A database now stores the list of files along with their various attributes such as author, assignment number, and timestamp.
- The server database remembers identities of files on other servers.
- Servers cooperate and keep replicated copies of a common database

Taking advantage of our experience, we changed the FX library a bit. Instead of storing an integer version number for the file, a hostname and timestamp were associated with it. This simplified establishing a version identity in an network of cooperating servers. Lists of files were returned as handles on linked lists rather than simple linked lists to ease storage management and passing of data over the network. We documented our new client library [Lewis1989a].

The basic operations were retained:

- send a file
- retrieve a file
- list files matching a template
- list access control list
- add to access control list
- delete from access control list.

The database is layered on *ndbm*. We rely on ndbm to allow an efficient scan of the entire database when we generate lists of files. Although a sequential scan of an entire database is slow, it is always faster than a find over a filesystem with the same number of nodes. So although this algorithm is slow, the perceived benefit is great. If very large courses are to be supported, this simple approach to database management can be replaced with a relational database which would allow matching on the various file parameters.

The database records information on the host responsible for holding the file, and there is a multi-server configuration that enables an authoritative database to be elected, and then shared among cooperating servers. The algorithms for electing and sharing are based on a simplification of the Ubik database system used in the Andrew Filesystem protection server.

The latest user interface

The original file exchange service used the GNU Emacs text editor with additional commands for annotating and displaying texts. The latest user interface integrates displaying, editing, formatting, exchanging, and annotating into two applications: *eos* for the student, and *grade* for the teacher. The student application gets its name from the Educational Online System, the term coined for the suite of programs and protocols specified for supporting the classroom activities. Here at last is one program, containing all the pieces, worthy of the name of the whole system. The teacher program gets its name from the command oriented grader program of the previous version of *turnin*.

These applications take advantage of the Andrew Toolkit (commonly called ATK) developed at Carnegie Mellon University [Palay1988a]. ATK offers the following features which made it particularly useful in building *grade* and *eos*:

- simple object oriented programming paradigm with inheritance to enable development by extending existing toolkit objects
- graphical user interface management system building blocks to support development of point and click applications
- rich selection of objects
- available on the X tape
- multi-font text object designed to look to the user like Emacs
- dynamic object loader for small initial application size, and simple expansion as the user uses advanced capabilities
- rapid application prototyping system.

Nowadays many people are developing graphical applications. Developing X based applications with object oriented toolkits has become de regueur. Availability on the X tape allows more freedom than proprietary development platforms, and a rich selection of objects increases the likelihood that the developer can use existing toolkit code rather than writing new code.

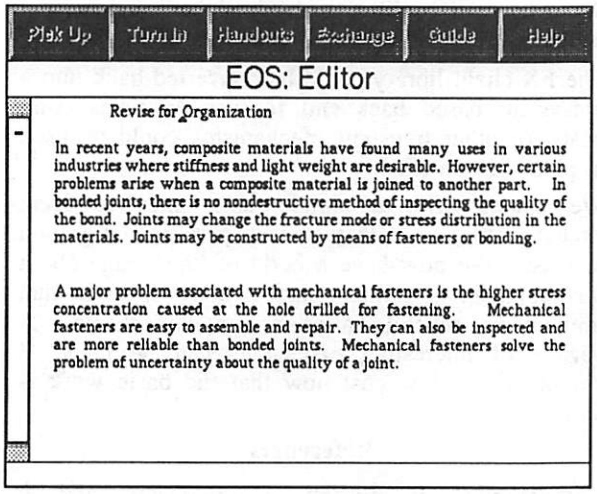
The multi-font text object was crucial. Virtually all multi-font text objects suitable for use as a stand alone text editor are extremely proprietary. We have not seen any other that has a code size under 1Meg. The ATK text editing objects are small, efficient, and extensible through the dynamic object loading system. We like being able to offer users the ability to edit equations, spreadsheets, and line drawings in *eos* without requiring all users to start up an *eos* containing all those subsystems. Until the time when students all run 12 MIP machines with 16 Meg of main memory, ATK offers an extremely attractive balance of size versus speed that no other toolkit has approached.

The original version of *eos* was done by N. Hagan Heller for her S. B. thesis at MIT in the department of Computer Science and Engineering [Heller1989a]. She used the ATK Development Environment Workbench (ADEW) to do her prototype. The ADEW system allowed her to compose and modify her screen layout directly with a WYSIWYG interface. Although we opted to re-implement her prototype in raw ATK toolkit calls, we could have used the ADEW code generator to build the application. We chose not to use ADEW because of some restrictions that ADEW imposed which have, for the most part been removed. Nick Williams, of Imperial College, University of London, visiting MIT for the summer re-implemented Hagan's design with assistance from me. Nick's implementation is outlined in an EUUG talk [Williams1990a].

The five student file exchange programs (*turnin*, *pickup*, *put*, *get*, and *take*), the editor, GNU Emacs, and the formatter (which was most often not used because it interfered too much with annotating) were made into an ATK editor with buttons across the top. Figure 2 shows *eos* with a typical short paper.

The teacher interface, *grade*, looks just like the student interface except that the *Turn In* and *Pick Up* buttons are replaced with *Grade* and *Return* buttons.

The *Guide* button opens a window on an on-line style guide that is being experimented with by the writing teachers who use eos. It replaces a GNU Emacs based on-line style guide that was too hard to use. The new one uses hyper-link buttons to access a whole lattice of information. The *Help* button is used to start the ATK *help* documentation browser.



EOS: Editor

Revise for Organization

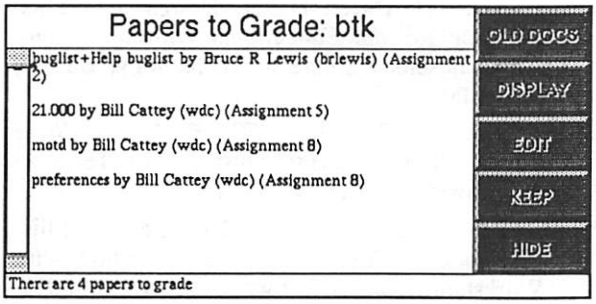
In recent years, composite materials have found many uses in various industries where stiffness and light weight are desirable. However, certain problems arise when a composite material is joined to another part. In bonded joints, there is no nondestructive method of inspecting the quality of the bond. Joints may change the fracture mode or stress distribution in the materials. Joints may be constructed by means of fasteners or bonding.

A major problem associated with mechanical fasteners is the higher stress concentration caused at the hole drilled for fastening. Mechanical fasteners are easy to assemble and repair. They can also be inspected and are more reliable than bonded joints. Mechanical fasteners solve the problem of uncertainty about the quality of a joint.

Figure 2: Screen dump of EOS student interface.

When a student clicks *Turnin*, a dialogue box pops up to get the filename and assignment number. The student is also given the choice of turning in the contents of the main editor window, or a file. In this way, users experienced with the old protocol of turning in a file will be able to use the new interface.

The *Grade*, *Handouts* and *Exchange* buttons all pop up a new window that contains a list of papers to select, along with appropriate operations to perform on the selected papers. For example, to annotate a paper turned in by a student, the teacher clicks the *Grade* button and positions the "Papers to Grade" window. Figure 3 shows a typical "Papers to Grade" window:



Papers to Grade: btk

buglist+Help buglist by Bruce R Lewis (brlewis) (Assignment 2)

21.000 by Bill Cattey (wdc) (Assignment 5)

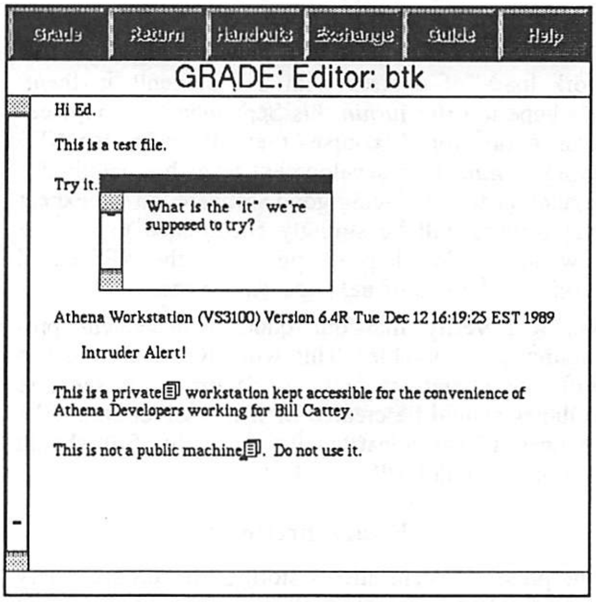
motd by Bill Cattey (wdc) (Assignment 8)

preferences by Bill Cattey (wdc) (Assignment 8)

There are 4 papers to grade

Figure 3: Screen dump of "Papers to Grade" window

The teacher clicks on the desired paper and then clicks the *Edit* button. The file is fetched from the *turnin* server into the main editor window of the grade program. The teacher then marks up the document. An object called *note* was developed for annotation. The ATK editor treats the *note* like a large character with internal state. When the *note* is **closed**, it appears as an icon of two little sheets of paper. When **open**, the text of the annotation is displayed. The user clicks on the icon to open the *note*, and on the black region at the top of the *note* to close it. Athena users of the ATK based editors such as *eos*, *grade* and *ez*, get the additional menu commands to create a new *note*, and to open and close all notes. When the teacher is finished modifying the file, he or she clicks the *Return* button, and the file is sent back to the *turnin* server for later *Pick Up* by the student. Figure 4 shows how a file with one open *note*, and two closed *notes* would look in a typical *grade* editor window:



GRADE: Editor: btk

Hi Ed.

This is a test file.

Try it.

What is the "it" we're supposed to try?

Athena Workstation (VS3100) Version 6.4R Tue Dec 12 16:19:25 EST 1989

Intruder Alert!

This is a private▤ workstation kept accessible for the convenience of Athena Developers working for Bill Cattey.

This is not a public machine▤. Do not use it.

Figure 4: Screen dump of active *grade* window

Although not shown in these basic demonstration figures, the ATK editor allows formatted text, and a rich variety of other types of data to be put in documents such as equations and line drawings. This capability will be exploited as more real-world documents such as proposals and articles are exchanged for annotation with *turnin*.

With the *grade* and *eos* applications, we finally realized the goal of enabling the teacher to point at papers on the screen, view them in final form, annotate them, and return them. The students are able to use the integrated system to receive the annotated papers, and use them directly for their next draft simply by deleting the annotations after reading them.

Deployment Strategy

We are following a conservative path for migrating users to the new server. Although the greater maintainability is tempting, our first priority is reliable service. We decided to extensively test and experiment with the new version before getting rid of the old version. Consequently the new server and applications programs have only been in use by two classes of 25 students each for the past term. The single server configuration has been running for 94 days so far without crashing. Nobody has reported a single problem with server reliability. These two courses, however can be considered well behaved. One of the professors had extensive experience with the old *turnin* and never tries anything out of the ordinary. The two courses share the same teaching assistant who also had extensive experience with *turnin*.

This summer we plan test *turnin* with simulated work loads of courses with 250 students in them. We hope to offer *turnin* this September as a replacement option for all courses presently using the NFS based *turnin*. Our development team has acquired a reputation for producing good software, so we expect that courses will be strongly encouraged to use the new *turnin*. We hope to phase out the NFS based *turnin* by the end of next academic year.

We will verify that our quota management procedures are workable. This will involve discussions with the operations staff to determine if separate volumes should be created or if the server should be changed to set a particular owner, or if it should enforce a quota itself.

Future directions

The present system allows storing files on secondary servers, identifying when all files are accessible, and merging in files from several places. It still has no provision for dividing work amongst servers in an equitable way. The list of servers to contact, and in what order is either registered with our *Hesiod* name server [Dyer1988a] , or set in the `FXPATH` environment variable. This makes determining primary and secondary servers a very static process.

Since the database is replicated, it should store a mapping of course name to a record of primary server and secondary servers. Then the FX library can contact any server for a list of the appropriate servers. The database can change the servers at any time. We initially expect a person to monitor the usage and adjust the database. In the far future heuristics to do load balancing automatically could be added.

We expect the user interface to continue to evolve into smoother integration with classroom procedures. We would like to produce a set of interfaces for industrial use. The user paradigm would be documents cycling between author and either management or peers for review and revision.

The command line user interfaces would work, as is, on any UNIX platform that could accommodate a port of the FX library. It is possible that the WYSIWYG integrated interfaces could be ported to other platforms. Since the Andrew Toolkit has window system independent imaging model, a Presentation Manager, or Macintosh Toolbox based system could be produced with only moderate effort.

The FX client library could be converted back into a filesystem based back end for use on timesharing hosts, or other transport mechanisms could be used in place of Sun RPC.

We started with a simple user interface idea: exchange files as if they were papers passed around in class. We now have a body of knowledge about this application domain, and a body of code that implements several variations on this theme. A variety of interesting extensions can be made at comparatively low cost now that the basic work is done.

References

Barrett1987a. E. Barrett, F. Bequaert, and J. Paradis, *Electronic Classroom: Specification for a user interface,* Athena Writing Project, Boston, MA (4 June 1987).

Camblor1988a. Maria Camblor, Rob Shaw, Bruce Lewis, and William Cattey, *Design Specification,* Athena Writing Project, Boston, MA (February 1988).

DellaFera1988a. C. Anthony DellaFera, ''The Zephyr Notification Service,'' *Proceedings of the USENIX Winter Conference*, pp. 213–219 USENIX Association, (February, 1988).

Dyer1988a. Stephen P. Dyer, ''The Hesiod Name Server,'' *Proceedings of the USENIX Winter Conference*, pp. 183–189 USENIX Association, (February, 1988).

Heller1989a. N. Hagan Heller, *Designing a User Interface for the Educational On-line System,* Massachusetts Institute of Technology, Boston, MA (May 1989).

Lewis1989a. B. R. Lewis, *File Exchange Client Library,* Massachusettes Institute of Technology, Boston, MA (June 1989).

Murman1988a. E. Murman, A. Lavin, and S. Ellis, ''Enhancing Fluid Mechanics Education with Workstation Based Software,'' *AIAA 88-0001*, (January 1988).

Palay1988a. Andrew. J. Palay, Wilfred J. Hansesn, Mark Sherman, Maria G. Wadlow, Thomas P. Nuendorffer, Zalman Stern, Miles Bader, and Thom Peters, ''The Andrew Toolkit - An

Overview,'' *Proceedings of the USENIX Winter Conference*, pp. 9–21 USENIX Association, (February, 1988).

Raeburn1989a. Ken Raeburn, Jon Rochlis, Stan Zanarotti, and William Sommerfeld, ''Discuss: An Electronic Conferencing System for a Distributed Computing Envi:onment,'' *Proceedings of the USENIX Winter Conference*, pp. 331–343 USENIX Association, (February, 1989).

Williams1990a. Nick Williams and William Cattey, *The Educational On-Line System,* EUUG (April 1990).

William Cattey has been at Project Athena since 1985, working primarily with the MIT Writing faculty on tools of general applicability such as *turnin* and *ez.* He is an Applications Programmer in the Educational Initiatives Group.

USENIX Summer Conference
June 11-15, 1990
Anaheim, California

Don Libes – National Institute of
Standards and Technology

expect: Curing Those Uncontrollable Fits of Interaction

ABSTRACT

UNIX programs used to be designed so that they could be connected with pipes created by a shell. This paradigm is insufficient when dealing with many modern programs that *demand* to be used interactively.

expect is a program designed to control interactive programs. **expect** reads a script that resembles the dialogue itself but which may include multiple paths through it. Scripts include:

- send/expect sequences – **expect** patterns can include regular expressions.
- high-level language – Control flow (**if/then/else, while,** etc.) allows different actions on different inputs, along with procedure definition, built-in expression evaluation, and execution of arbitrary UNIX programs.
- job control – Multiple programs can be controlled at the same time.
- user interaction – Control can be passed from scripted to interactive mode and vice versa at any time. The user can also be treated as an I/O source/sink.

expect successfully deals with interactive programs. It also solves several other large classes of problems which UNIX shells do not.

Keywords: expect, interaction, programmed dialogue, shell, Tcl, UNIX, uucp

1. Introduction

UNIX programs used to be designed so that they could be connected with pipes created by a shell. This paradigm is insufficient when dealing with many modern programs that *demand* to be used interactively.

For example, the **passwd** program is used to change passwords. **passwd** prompts for the password. There is no provision for passing the information any other way. This means that you cannot write a shell script which uses **passwd** without letting it do the prompting and reading. Thus, it is impossible to write a script that, say, rejects passwords that are in the system dictionary.[1]

This illustrates one type of difficulty in the user interface provided by shells such as **sh**, **csh**, **ksh** and others (which I will generically refer to as ‘‘the shell’’ in the rest of the paper). I will discuss several other difficulties later. All of them have to do with the shell's inability to communicate with interactive programs. My solution is called "**expect**".

expect is a program that "talks" to other interactive programs according to a script. By following the script, **expect** knows what can be expected from a program and what the correct responses should be. An interpreted language provides branching and high-level control structures to direct the dialogue. In addition, the user can take control and interact directly when desired, afterward returning control to the script.

The name "expect" comes from the idea of send/expect sequences [4] popularized by **uucp**, **kermit** and other communications programs. However, unlike these programs, **expect** is generalized so that it can be run as a user-level command with any program and task in mind. (**expect** can actually talk to several programs at the same time.)

Using **expect,** it is possible to create a script that solves the **passwd** problem. Here are some other things **expect** can do, each requiring only a small amount of script:

- Have your computer dial you back, so that you can login without paying for the call.
- Start a game (e.g., **rogue**) and if the optimal configuration does not appear, restart it (again and again) until it does, then hand over control to you.
- Run **fsck**, and in response to its questions, answer "yes", "no" or give control back to you.
- Connect to another network or BBS (e.g., MCI Mail, CompuServe) and automatically retrieve your mail so that it appears as if it was originally sent to your local system.

Although these problems are conceptually simple, none of them can be solved by the shell. What is wrong? What do we do with the hard cases!?!

[1] Ironically, the original reason for having **passwd** perform the prompting was for security!

2. Solving the problem

Since I am claiming that **expect** can solve problems the shell cannot, let us first start by reviewing what the shell is capable of.

Each process created by the shell is given a standard input (stdin), standard output (stdout) and standard error (stderr) (although I will ignore the last one for now). The shell can connect these to other processes or files. However, shell pipes and redirection are purely one-way. Ritchie [7] has described shell pipe notation as "*unabashedly linear*". There is no shell notation to create two processes which have their standard input and output cross-connected (Figure 1).

The traditional notation *proc1* | *proc2* indicates that output flows from *proc1* to *proc2*. There is no data flow from *proc2* to *proc1*. Indeed, the only entity that the shell can interact with on a 2-way basis is the user. Viewed from the opposite direction, only the user is capable of 2-way interaction with programs.

Since the shell cannot construct such connections nor can it participate in them, it cannot "talk" to interactive programs. This prevents any program from interacting with another unless both have been specially designed to do so. This is the first problem.

A second problem is that the shell has no way to prevent a program from bypassing the standard input and output conventions. Programs are free to open **/dev/tty** to communicate directly with the user. This is often used to bypass shell redirection. For example, **crypt** does this because its input is redirected while it interactively demands an encryption password.[2]

The first problem is easy to solve. A program is created (by the shell) which internally spawns the process to be controlled. The spawned process is established so that its standard input and standard output remain connected to the original program (Figure 2).

The parent now reads a command script. When it finds **send** commands, it sends data to the child. When it finds **expect** commands, it watches the output of the child for a pattern. If the pattern is matched, the parent goes to the next command in the script (Figure 3).

When an **interact** command is found, the parent simply copies characters from the user to the child and vice versa (Figure 4). Control may be returned to the script at the user's convenience.

These simple ideas are the heart of **expect**. However, a few refinements allow **expect** to address several more problems.

2.1 Pseudo-terminals

Pipes do not support terminal semantics. For example, programs such as **rogue**, **emacs**, etc., which require the terminal size in order to run, will not run over pipes. Thus, **expect** uses pseudo-terminals (ptys). Ptys are logical device drivers that give the "look and feel" of real terminal drivers despite the fact that they are used to connect two processes together. In addition, ptys solve the **/dev/tty** problem noted above. Programs that open **/dev/tty** will actually end up speaking to their pty.

Ptys support two paths of communication flow (much like two pairs of pipes). What happens to the standard error? It is overloaded into the path already used by standard output. The rationale is that if a user sitting at a terminal can make decisions based on a common output stream of both standard output and the standard error, then so can the script.

[2] Again, this was done for security reasons.

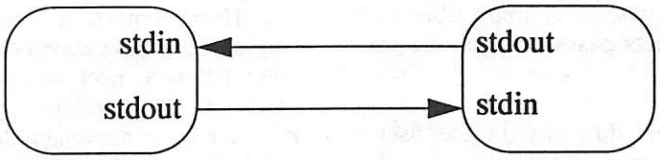


Figure 1. The shell cannot connect two processes in this way.

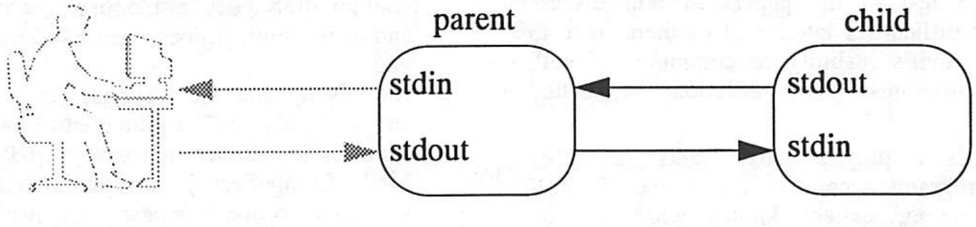


Figure 2. The parent creates a child process so that it can read the child's standard input and write its standard output. The user is left connected by the shell, but plays no role here.

Just as the user may redirect either standard input or standard output, so may the script. But by default, neither is redirected. This is true for both **expect** and the shell.

2.2 Job control

The script may interact with a number of processes simultaneously. Figure 5 shows several processes being controlled at the same time. To control multiple processes, a user would use job control. So does **expect**. Script job control is actually easier to use because it can be programmed whereas shell job control must be hand entered.

For example, suppose you are using **csh** and have to type something at *proc2* depending upon what *proc1* is telling you. If this happens 100 times, you have to type **^Z/fg** sequences 200 times![3] With **expect**, a simple loop suffices. A good example is to try and connect two Eliza [10] or **chess** processes to each other. Remember that the output of the UNIX **chess** program is not directly usable as input.

The user can also be manipulated as if they were a process. The effect is exactly like a shell script reading from and writing to the user. In other words, the source of I/O to the user is the script rather than an underlying interactive process. This is illustrated by the user appearing alongside the processes.

If desired, the user can take control (appearing on the left side of the figure) and enter commands just like the script. Both the script and the user can take control from and return it to each other. In the figure, the two have been moved closer together to emphasize the near equal relationship of them.

2.3 High-level language

The last important feature of **expect** is the language itself. It is described in the next section.

3. expect scripts – What do they look like?

expect scripts are written using a high-level procedural language. The language is interpreted and resembles the shell in many ways. Elements are also derived from C and LISP. Despite its mixed heritage, much of the excess baggage from these other languages has been omitted leaving a modest but capable language. The language consists of a core of features called *Tcl* (Tool Command Language) and is described by Ousterhout [5]. This section will only give a brief overview and enough details to describe the sample **expect** scripts later on.

[3]Even if you were using a window system, you would still have to cut and paste 100 times.

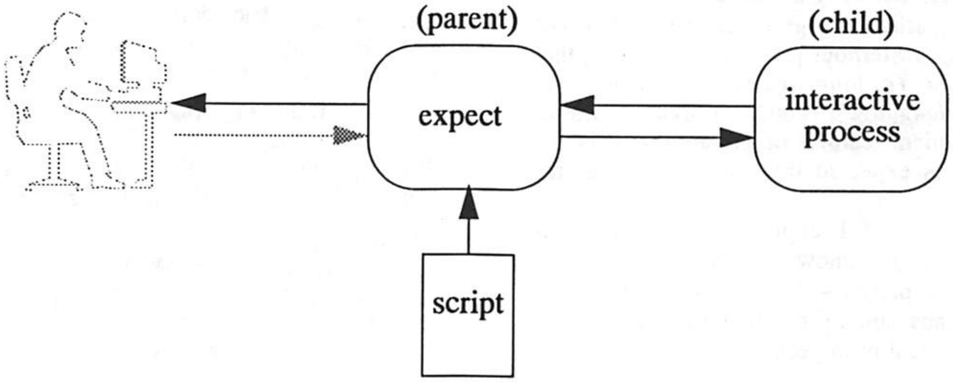


Figure 3. **expect** "talks" to the child process, according to the script. Interaction is copied to the user terminal and appears as if the user actually typed it.

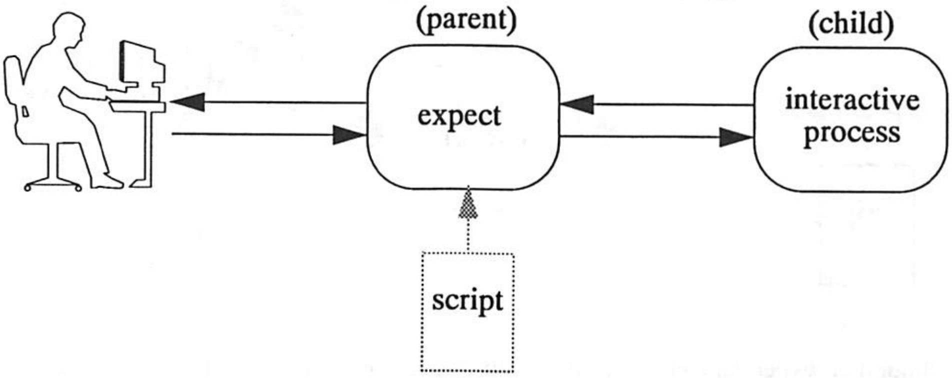


Figure 4. In **interact** mode, the user takes control and types directly to the child process.

The Tcl core consists of control flow statements such as **if, while**, and **case**. Tcl supports procedure definition, recursion, scoping, and more. UNIX programs may be called and files manipulated. Expression evaluation is provided by a small set of primitives that manipulate the only type – strings. (Conversion to and from other types is performed automatically, a la SNOBOL.)

The following Tcl fragment (from [5]), swaps the values of variables **a** and **b**, if **a** is less than **b**.

```
if {$a < $b} {
    set tmp $a
    set a $b
    set b $tmp
}
```

Here is a command to define a recursive factorial procedure:

```
proc fac x {
  if {$x == 1} {return 1}
  return [expr{$x * [fac[expr $x-1]]}]
}
```

The syntax and semantics are sufficiently close to C and the shell that the meaning of these examples should be intuitively obvious. For lack of space, I will not describe Tcl further, however it is completely described by Ousterhout [6]. For that matter, it is not particularly germane to the theory of **expect**. Indeed, Ousterhout [5] makes the point that the "*syntax of the Tcl language is unimportant: any programming language*" could provide similar features. The salient features of Tcl are that it is:

- simple – It is expected that most Tcl programs will be short.
- programmable – Tcl applications are general-purpose and are not known in advance.
- efficiently interpreted – Tcl must be able to execute commands quickly enough that user interaction is not noticably impeded.
- internally interfaceable to C – Tcl must allow one to add new commands that work synergistically with existing Tcl commands.

As the last bullet says, Tcl is designed to allow the addition of new commands. **expect** adds twelve commands to the Tcl language. I will now present the more interesting of these new commands.

3.1 Interaction commands

send *args*
sends *args* to the current process. Strings are interpreted following Tcl rules. For example, the command

```
send hello world\r
```

sends the characters, h e l l o <space> w o r l d <return> to the current process.

expect *patlist1 action1 patlist2 action2 . . .*
waits until a pattern matches the output of the current process, or a specified time period has passed. Each *patlist* consists of a single pattern or list of patterns. If the pattern matches, the corresponding action is executed. The result of the action is returned from **expect**. The exact string matched (or read but unmatched, if a timeout occurred) is stored in the variable **expect_match**. If *patlist* is **eof** or **timeout**, the corresponding action is executed upon end-of-file or timeout, respectively. The default timeout period is 10 seconds but may, for example, be set to 30 by the command `set timeout 30`.

The following fragment is from a script that involves a login. **abort** is a procedure defined elsewhere in the script, while the other actions use Tcl primitives similar to their C namesakes.

```
expect \
   {*welcome*} break \
   {*busy*}    {print busy; continue} \
   {*failed*}  abort \
   timeout     abort
```

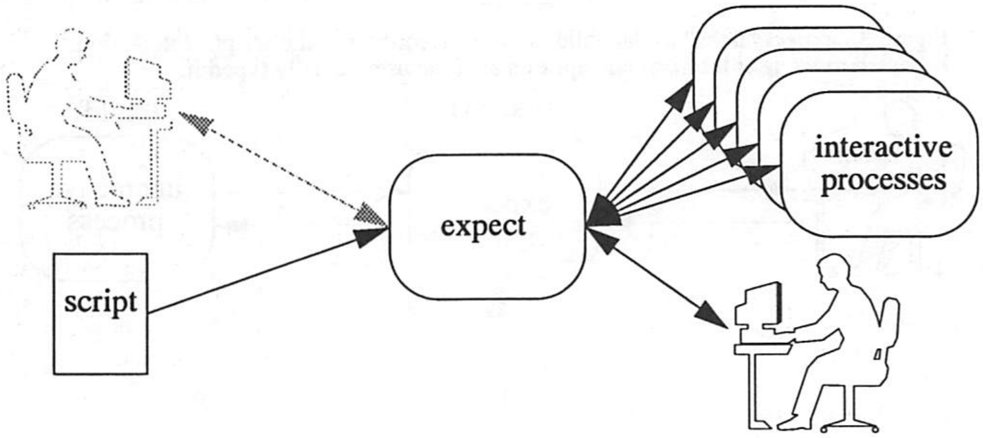


Figure 5. **expect** is communicating with 5 processes simultaneously. The script is in control and has disabled logging to the user. The user only sees what the script says to send and is essentially treated as just another process.

Patterns are the usual C-shell-style regular expressions. Patterns must match the entire output of the current process since the previous **expect** or **interact** (hence the reason most are surrounded by the * wildcard). However, more than 2000 bytes of output can force earlier bytes to be "forgotten". This may be changed by setting the variable **match_max.**

interact *[escape-character]*
gives control to the user. User keystrokes are sent to the current process, and the standard output and standard error of the current process are returned to the user. Any valid script commands may be entered after pressing the optional *escape-character*. Control is returned to **interact** if the continue command is entered. If the **return** command is entered, **interact** immediately returns with the argument of **return** (or the empty string if none is given).

During **interact** (except when entering commands via the escape character), job control is disabled so that all characters may be passed to the current process.

3.2 Job control commands

close
closes the connection to the current process. Most interactive programs will detect EOF on their standard input and exit; thus **close** usually suffices to kill the process as well. Both **expect** and **interact** will detect when the current process exits and implicitly do a **close**.

spawn *program [args]*
creates a new process running *program args*. Its standard input, standard output, and standard error are connected to **expect**, so that they may be read and written by other **expect** commands. The connection is broken by **close** or if the process itself closes any of the file descriptors.

When a process is started by **spawn**, the variable **spawn_id** is set to a descriptor referring to that process. The process described by **spawn_id** is considered the *current process.*

spawn returns the UNIX process id. Note that this is not equivalent to the descriptor in **spawn_id**.

Internally, **spawn** uses a pty, initialized the same way as the user's tty. When this is not possible (such as when **expect** has no controlling terminal), **spawn** uses the default pty settings. If these are not appropriate, the user can **spawn** a shell, set the pty parameters directly, and then **send** (rather than **spawn**) the original command to the shell.

select *spawn_id1 spawn_id2 ...*
returns a subset of the given *spawn_ids* that have input pending. **select** waits until at least one *spawn_id* can be read or until the timeout (see **expect** command above) has expired.

There is no explicit command to switch jobs. Rather, the variable **spawn_id** determines the current process. **spawn** sets this as a side-effect so that a script interacting with only one process need not ever mention **spawn_id**. **spawn_id** may be read and written through Tcl's set command.

Here is an example showing how job control could be used to have two **chess** processes interact. After spawning them, one move is sent by hand to get things started. In a loop, a move is sent from one process to the other, and vice versa. The **read_move** and **send_move** procedures are left as an exercise for the reader. (They are actually very easy to write, but too long to include here.)

```
spawn chess
set chess1 $spawn_id
spawn chess
set chess2 $spawn_id
# force someone to go first
send p/k2-k3
for {} {1} {} {
    read_move
    set spawn_id $chess1
    send_move
    read_move
    set spawn_id $chess2
    send_move
}
```

There is no command (analogous to **csh's bg**) to let processes run without interaction, since presumably input is a necessary part of interaction and cannot be supplied in the background. Processes that enter into a lengthy phase during which no input takes place will free run by default, although output will eventually clog the pty if not periodically flushed. Fortunately, this is easy to do.

3.3 Miscellaneous commands

The remaining commands will not be described in detail. For complete descriptions, see Libes [2]. These miscellaneous commands fall into the following classes:

- tracing – Programs may be traced to assist debugging.
- exiting – **expect** can return an exit code, allowing it to be intelligently used in shell scripts.
- logging – Logging to files and/or the user terminal is flexible. This allows interaction with the user while hiding some or all of the interaction taking place with programs.

4. More examples

expect scripts are similar in style to shell scripts. They look like the interaction they are supposed to control. Just as shell scripts primarily consist of the commands as a user might type them,

expect scripts consist primarily of the interaction that a user might see.

Here are two examples. The first runs the BSD adventure game **rogue** repeatedly until a configuration with unusually good attributes (i.e., strength of 18) appears, after which control is given to the user.

```
# rogue.exp – find a good rogue game
set timeout 3
for {} 1 {} {
    spawn rogue
    expect {*Str:\ 18*} break \
           timeout      close
}
interact
```

Some comments are in order: The first line is a comment, naming the file and explaining what it does. The second line sets a short timeout. This is appropriate since we are dealing with a local program that will respond very quickly.

for introduces a C-like **for** loop, with the same control arguments as in C. Here, the loop repeats forever. After **rogue** is started, we look for the text of interest in the output. **rogue** is a graphics program which uses curses. Curses does not guarantee screens are created in an intuitive manner, and **expect** programmers must understand that. However, it is not a problem here.

If **expect** does not find what it is looking for, the dialogue is terminated via **close** (which will cause **rogue** to go away) and the loop restarted. If the desired string is found, the loop is terminated and the user given control of the dialogue through **interact** on the last line.

The second example dials a phone. It can be used to reverse the charges, so that long-distance phone calls are charged to the computer. It is invoked as `expect callback.exp 12016442332`; the script is named **callback.exp** and +1 (201) 644-2332 is the phone number to be dialed. (Scripts may also be turned into executables on systems which support the **#!** magic.)

```
# give user some time to logou
exec sleep 4
spawn tip modem
expect {*connected*} {}
send ATZ\r
expect {*OK*} {}
send ATDT[index $argv 1]\r
# modem takes a while to connect
set timeout 60
expect {*CONNECT*} {}
```

The second line illustrates how a UNIX command with no interaction can be called. **sleep** 4 will cause the program to block for four seconds, giving the user a chance to logout, since the modem will presumably call back to the same phone number that the user is already using.

After spawning **tip**, the modem dials the number. (The modem is assumed to be using Hayes protocol, but it would be easy to expand the script to handle others.) No matter what happens, **expect** terminates. If the call fails, it is possible for **expect** to retry, but that is not the point here. If the call succeeds, **getty** will detect DTR on the line after **expect** exits, and prompt the user with `login:`. (Actual scripts usually do more error checking.)

This script illustrates the use of command-line parameters, made available to the user as a list named **argv**. Commands may also be passed in and executed by the program. A special flag (**-c**) allows execution of commands before any in the script. For example, an **expect** script can be traced without reediting by invoking it as `expect -c "trace ..." script.exp` (where the ellipsis indicates a tracing option).

5. What classes of problems does expect address?

expect addresses a surprisingly large class of problems that the shell does not. At the same time, **expect** does not attempt to subsume functions already handled by other utilities. For example, there is no built-in file transfer capability, because **expect** can just call a program to do that.

The following categories are not meant to be disjoint but to sharpen the focus of examples that may share multiple problems.

5.1 Programs that demand interactivity

Programs that demand interactivity such as **passwd** and **tip** are easy to control with **expect**, but impossible with the shell. In fact, **tip** has special code to do dialing before a conversation, but it is quite limited in power. **expect** eliminates the need for this type of special code in many programs.

5.2 Programs that cross machine or program boundaries

Making a shell script run across machine boundaries is not possible except in limited ways. For example, shell scripts that involve **telnet**ing to another host cannot log in nor can they continue the shell script on the remote host. **expect** does not see these kinds of machine or program boundaries.

5.3 Programs that read and write /dev/tty

Programs that read and write **/dev/tty** cannot be used from shell scripts without the shell script accessing **/dev/tty**. **passwd**, **crypt**, and **su** are examples of programs that cannot be controlled by the shell but can by **expect**.

5.4 Programs that flush input

Some interactive programs believe they are doing the user a favor by flushing input after detecting an error. Particularly clever programs such as **rn** [9], not only flush input already received but continue to flush input for a short time afterwards to allow for communications or user delays.

Redirecting standard input from the shell is ineffective with such programs since there is no control over how much can be lost when input flushing occurs. **expect**, on the other hand, will wait for the desired prompt rather than proceeding to send commands blindly.

While it was not my intention, **expect** provides a foundation for a relentless password cracking tool. However, to show my compassion I will remind system adminstrators that one preventative measure is to lock out an account after a small number of incorrect passwords have been tried.

5.5 Passing control from user to/from script

Programs such as **rogue, tip, telnet**, and others have a frequently repeated, well-defined set of commands and another set that are not well-defined. For example, **telnet** is always started by logging in, after which the user can do anything. **expect** can pass control from the script to the user to provide this ability. In fact, **expect** can take control at any time to execute sequences of commonly repeated commands.

5.6 Ostensibly non-interactive programs

Many programs are ostensibly non-interactive. This means that they can be run from a shell script, but with greatly diminished functionality. For example, **fsck** can be run from a shell script only with the **-y** or **-n** options. The manual [1] defines the **-y** option as follows:

> "*Assume a yes response to all questions asked by fsck; this should be used with extreme caution, as it is a free license to continue, even after severe problems are encountered.*"

The **-n** option has a similarly worthless meaning. This kind of interface is inexcusably bad, and yet many programs have the same style. For example, **ftp** has an option that disables interactive prompting so that it can be run from a script. But it provides no way to take alternative action should an error occur.

expect is useful with such programs. For example, it could be programmed to answer "`yes`" and "`no`" depending on the question from **fsck**. Control could be turned over to the user for questionable cases.

5.7 Programs with poorly written interfaces

Ousterhout [5] makes the observation that "*a general purpose, programmable command language amplifies the power of a tool by allowing users to write programs in the command language in order to extend the tool's built-in facilities.*"

Few tools actually include such a language. Examples are shells and **emacs**. There are also a few that have more simplistic facilities such as the **.rc** files of **mail, vi**, and **dbx**. But in all, there are very few tools with the flexibility of a good language. Indeed, Tcl was designed to address this very problem.

Expecting UNIX tools to be rewritten using Tcl is a noble but unlikely proposition. However, a similar effect can be achieved with **expect**. For example, **expect** can be used to initialize a tool, much like a set of commands from an **rc** file. And like Tcl, **expect** presents a uniform language for doing so. In effect, **expect** provides a way of giving the power of Tcl to tools without any effort spent rewriting them.

If desired, **expect** can be run in the background, completely disassociated from user input. **expect** is capable of returning a status value to a script, often more meaningfully than the original tool or task. (Realistically, there is little reason for a program that originally could only have been run interactively to return a status of any type.)

5.8 Multiple programs never designed to work together

expect is capable of connecting programs that were not originally designed to be connected. In contrast to non-interactive filters that form pipelines, interactive programs have foresaken any attempt to be driven by another program. Eliza and **chess**, both mentioned earlier, are good examples.

A more complex example is communication with another network or bulletin board system. Commercial systems such as MCI Mail and CompuServe do not forward mail, expecting that users will dial up and read mail interactively. An **expect** script can dial up such a system and check for mail. If mail is found, a **mail** process can be started on the local system and fed input from the remote system. Mail will then appear as if it was originally mailed to the local system. Since **expect** can run in the background, this can be done at night, every hour, or whatever is convenient.

5.9 Dynamic and complex pipes and redirection

A number of projects have been built to step beyond the linearity of pipes enforced by the shell. Two notable examples are **gsh** and MTX.

gsh [3] is based on the Bourne shell, but handles graphs of processes, such as sending the output of one process to two processes, or building a set of three process in a cycle. While **expect** was not designed for this purpose, it can do this as a byproduct of its complete control of any dialogue. Of course, the result will not be as fast because **expect** necessarily interposes itself in order to control the dialogue.

MTX [8] is a screen-based pipe manager. It solves the same set of problems as **gsh**, although the interface is mouse-oriented instead of keyboard-oriented. In addition, MTX can rearrange connections in use. It does this using the same pty mechanism that **expect** does (with a similar penalty in throughput), although MTX does not provide any automated control of the dialogue.

In summary, **expect** can emulate dynamic and complex pipes and redirection. It is a simple matter to emulate processes of pipes in a graph. Automatic rearrangement is possible either under the control of a user or when signalled by data. Complex redirection such as arbitrary fan-out is also trivial and easily supercedes the capabilities of **tee**.

6. Is expect a shell?

The beginning of this paper compared the shell to **expect**. To repeat, the shell is incapable of the interactive dialogue that **expect** can perform. But how does **expect** compare with the shell? Is **expect** as powerful as a shell?

The base language of **expect**, Tcl, is quite powerful and is certainly capable of doing the same kinds of programming as the shell. Indeed, as a programming language, Tcl is functionally almost a replacement for any of **sh**, **csh** and **ksh**. One noticeably missing feature is redirection, which Tcl can perform by calling the shell.

For interactive use, however, Tcl has little support. There is no history or job control. This is understandable, considering that Tcl was not designed to be used interactively. The additional commands added by **expect** do not change that. Job control as supported by **expect** is strictly command-based while the shell offers interrupt keys as shorthand. Indeed, typing job control characters at **expect** itself only affects **expect**, not the other processes **expect** is interacting with. (The exception to this is that when in **interact**, job control characters are sent to the current process.)

While **expect** can be used interactively (by pressing the *escape-character* while in **interact**), it was not designed to be and lacks pleasant features such as history and interrupt key-based job control. Its interactive mode is seen as primarily useful for experimenting, although some very powerful results are possible by typing **expect** commands in directly, or indirectly by say, programmable function keys or software stream modules that perform interactive line editing.

expect can call the shell and be called by it. The two work very well together. An obvious question is if one can be subsumed into the other. Adding the **expect** commands to the shell is probably the easiest implementation, and I see little technical difficulty in doing so.

7. Implementation discussion

This section discusses areas of the implementation that are unusual in some way and might be educational to a large percentage of readers. The source code is quite readable and perusal of it is encouraged.

7.1 Command language

From the list of examples in section 5, it should be apparent that the functions provided by **expect** have long been desired. I began thinking about them several years ago and have experimented with various implementations. The biggest stumbling block was the language. I knew I needed one, I knew how to write one, but I wasn't sure how far to go nor was I particularly interested in the task of writing yet-another-utility-language.

Tcl was the solution: it was designed specifically as an embeddable language. Tcl comes with a core of commands to which the application writer can add application-dependent commands. Adding the **expect** commands was relatively painless although a number of different command designs were tried before being finalized.

In my environment (Sun 3 running Sun OS 4.0.3), the Tcl library (version 2.1) is approximately 8000 lines, including comments (45k object code); the additional **expect** source (version 1.7) is 1700 lines (13k object code). Clearly, the Tcl code dominates **expect**. Here, **expect** is a wrapper around Tcl, which is probably different than how the original Tcl designers foresaw its role.

Tcl is not the only possible base for **expect**-like functionality. Prior to Tcl, I looked at the send/expect control used by **uucp**, **kermit**, and other communication programs. However, these are quite primitive and do not even provide adequate flexibility for their own tasks. For example, system administrators always embed calls to **uucp** in shell scripts which can repeat dialing upon failure.

Two alternatives to Tcl that I could have chosen are **emacs** and **perl**. While not specifically designed to provide a language for tools, both systems have embedded interpreters that can be used this way. Unfortunately, the tradeoffs are numerous and deserve more space than I can devote here. It would probably be worth doing an implementation in each to compare them.

7.2 Multisource asynchronous I/O

expect requires multisource asynchronous I/O primarily for the **interact** command (which listens for characters from the user and a process at the same time). Using this feature requires different implementations on different UNIX systems is probably the least portable part of the software.

Berkeley UNIX has long supported the **select** system call which permits waiting for activity from a set of file descriptors. Thus, virtually all BSD-derived UNIX systems support **select**. On the other hand, System V has only recently supported an equivalent of **select**. Release 3 supports **poll** which is comparable to **select**; however, a large percentage of SV systems are still based on Release 2, which forces applications to poll by busy-waiting. (POSIX has not yet provided a means of performing multisource asynchronous I/O, though it seems inevitable.) Earlier systems, such as V7, could not even **poll**. Typically, programs such as **uucp** forked an auxiliary process which blocked waiting for data from one direction while the original process blocked waiting for data from the other direction. I consider this untenable, since some **expect** applications spawn many processes, both sequentially and in parallel. Using this style of communication requires two extra utility processes for each real process requested by the user. For example, Figure 5 would need 12 more processes than it does in the current implementation.

For lack of more sophisticated code (and being not clear that it is even possible), the implementation cannot simultaneously interact with multiple processes without **select**, **poll**, or something similar.

7.3 Job control

This section discusses the interaction between BSD-style job control and **expect**. **expect** has its own way of controlling jobs, discussed extensively in section 2.2.

In a sense, **expect** finesses the problem of job control. When sitting at the keyboard running **expect**, a user may perform job control (i.e., by pressing job control characters), which will affect the **expect** program just like any UNIX program. For example, by default, a ^Z will stop **expect**.

An exception to this is during the **interact** command. Since **expect** has no idea if its client programs are interested in seeing job control characters, all characters (except an optional escape character) are passed through to the current process. Thus, programs that run in raw mode (e.g., **rogue**) and programs that handle job control themselves (e.g., **csh**) can run with their full functionality.

Of course, users choosing to interact directly must understand that programs which do not handle job control signals can be unpleasant. For example, sending a ^Z to a program that does not catch SIGSTOP will cause it to be stopped by the kernel. But since there is no program behind it to catch control, **expect** will wait as well. The program can be resumed by sending a SIGCONT, but this is presumably inconvenient. The moral is, when dealing with programs that do not understand job control, either do not send them job control signals or place a shell behind them that does.

Switching jobs internal to **expect** from within **interact** can be done without needing to back every process with a shell. The technique is to escape to the **expect** interpreter (by pressing **interact**'s escape character), set **spawn_id** to the desired process, and return to **interact** (via **return**). The user will now be interacting with the desired process.

7.4 Throughput

While throughput statistics are *de rigueur* in USENIX implementation papers, it is difficult to quantify performance in this type of program. For example, how does one compare **expect**'s overall impact on system throughput to that introduced by canonical input processing of typical human typing? About the only thing that is clear is that **expect** uses a fraction of the real time that a user does.

Internally, **expect** processing is heavily dependent upon scripts. For example, the **rogue** script presented earlier examines about 10 games per second (and is fun to watch). Most of the real time is spent waiting for the game itself. Of the CPU time, about 40% is spent pattern matching to guide the script, 26% in I/O, 16% in **open**, **close**, and **ioctl**, 8% in **fork**, and 5% in timer calls. The large amount of time spent in **open**, **close**, and **ioctl** are due to the inefficient technique required by BSD UNIX to locate and initialize ptys.

The **rogue** script is so biased towards short dialogues, it is likely that most other scripts will spend even larger percentages of time pattern matching. While guiding the dialogue is a primary function, this indicates an area for improvement. The current (Tcl-supplied) pattern matcher could be improved, for example, by compiling patterns. But **expect** has even more demands. Regular expression pattern matching is performed on input each time a **read** completes. If characters arrive slowly, the pattern matcher scans the same data many times. System indigestion can play a big role here, as larger scheduling quanta drive the pattern matcher less frequently when more characters at a time are gathered from each pty. The performance of a pattern matcher that does not need to rescan over earlier data needs to be studied.

8. Comments and conclusions

The UNIX shell paradigms are incapable of intelligently managing interactive programs. This has been a long-standing problem, traditionally solved by avoidance. Yet the number of interactive programs grows daily, and shells have not changed to address this.

expect solves these problems directly and with elegance. **expect** scripts are small and simple for problems that are small and simple. While I am not so naive to believe all **expect** scripts will be small, it is apparent that the scripts scale well. They are comparable in style to shell scripts, being task-oriented, and provide synergy with shell scripts, both because they can call shell scripts and be called by them. It would be a worthwhile experiment to marry the features of **expect** to the shell, and I see little technical difficulty in doing so.

Some interesting open questions remain: How would the buffering work in a combined **expect/select** command. If **expect** had a built-in terminal emulator, could one look for "regions" of character graphics? Lastly, how could **expect** emulate interactions with window systems, such as mousing and dragging. Each of these requires further research. Nonetheless, as long as there are shells, there will be interactive programs that are not controllable by them, and **expect** will continue to be useful.

9. Acknowledgements

This work was jointly funded by the NIST Automated Manufacturing Research Facility (AMRF, project 734-3385) and Scientific and Technical Research Services (STRS, project 734-3106).

Thanks to Scott Paisley for writing a program called **stelnet** (smart **telnet**). **stelnet** ran **telnet** and performed a simple send/expect conversation to login. **stelnet** had only straight-line control without error processing, used pipes instead of ptys, and lacked pattern matching and job control. Nonetheless, **stelnet** proved very useful in the AMRF, and got me to thinking about a more generic tool.

John Ousterhout is responsible for Tcl, without which **expect** would not have been written. John also critiqued **expect** as well as this paper. I am indebted to him.

Several people made important observations or wrote early scripts while I was still developing the command semantics. Thanks to Rob Densock, Ken Manheimer, Eric Newton, Scott Paisley, Steve Ray, Sandy Ressler, and Barry Warsaw. And also, like, thanks to my grammarians, Scott Bodarky, Ted Hopp, and Sue Mulroney, who read read and corrected everry sentence in the paper but like this one, right.

10. Availability

Since the design and implementation of **expect** was paid for by the U.S. government, it is in the public domain. However, the author and NIST would like credit if this program, documentation or portions of them are used. **expect** may be **ftp**'d as **pub/expect.shar.Z** from **durer.cme.nist.gov**.

11. References

[1] AT&T, UNIX Programmer's Manual, Section 8.

[2] Don Libes, "*expect(1) – programmatic dialogue with interactive programs*", unpublished manual page, National Institute of Standards and Technology, February, 1989.

[3] Chris McDonald and Trevor Dix, "Support for Graphs of Processes in a Command Interpreter", *Software: Practice & Experience*, Volume 18 Number 10, p. 1011-1016, October 1988.

[4] D. Nowitz, "*Uucp Implementation Description*", UNIX Programmer's Manual, Bell Laboratories, October, 1978.

[5] John Ousterhout, "Tcl: An Embeddable Command Language", *Proceedings of the Winter 1990 USENIX Conference*, Washington, D.C., January 22-26, 1990.

[6] John Ousterhout, "*tcl(3) – overview of tool command language facilities*", unpublished manual page, University of California at Berkeley, January 1990.

[7] Dennis Ritchie and Ken Thompson, "The UNIX time-sharing system", *Communications of the ACM*, Volume 17, Number 7, 635-375 (1974).

[8] Stephen Uhler, "MTX – A Shell that Permits Dynamic Rearrangement of Process Connections and Windows", *Proceedings of the Winter 1990 USENIX Conference*, Washington, D.C., January 22-26, 1990.

[9] Larry Wall, "*rn(1) – read news program*", unpublished manual page, May 1985.

[10] Joseph Weizenbaum, "Eliza – A Computer Program for the Study of Natural Language Communication between Man and Machine", *Communications of the ACM*, Volume 9, Number 1, January 1966, p. 36-45.

Don Libes received a B.A. in Mathematics from Rutgers University and an M.S. in Computer Science from the University of Rochester.

Currently at the National Institute of Standards and Technology, Don is engaged in research that will help U.S. industry measure the standard hack. Unfortunately, NIST does not have a very good sense of humor, so he was forced to write his first book "Life With UNIX" through Prentice-Hall.

USENIX Summer Conference
June 11-15, 1990
Anaheim, California

Simon Kenyon – ICL Information Technology Centre

IAW – The Intelligence Analyst Workbench

ABSTRACT

IAW is a graphical front-end to a criminal intelligence data base that has been constructed by the Royal Hong Kong Police (RHKP), using INDEPOL, an ICL data base system which runs on our mainframe range of computers. It is a tool to help analysts make sense of large quantities of intelligence data by presenting it to them in an easily digestible graphical format. The methodology embodied in IAW is based on work done by Anacapa Sciences Inc., a company which specializes in human factors research. They have developed techniques to graphically interpret intelligence data.

Not only is IAW an interesting application in its own right, but the way in which we have tackled its development is worthy of mention. We have implemented it using SEPIA, a prolog system which has an interface to PCE, an object-oriented graphics system. IAW runs on a Sun workstation running the UNIX[1] Operating System and using X11 Release 4 and the OPEN LOOK[2] graphical user interface.

The choice of a declarative language like prolog, coupled with an object-oriented graphics system, has proved to be extremely flexible. It has resulted in a system which has been faster to develop and more amenable to change. In addition, because we have made the system internally very modular, we have left the way open for code reuse in other applications.

Introduction

INDEPOL is an ICL database which runs on our Series 39 range of[1986a] mainframes. One of the goals of the original designers of INDEPOL was to make it as general purpose as possible. It was originally developed for the UK Ministry of Defence for a particular application but it has since been put to other uses; particularly in the defence and police areas. Using INDEPOL, the Royal Hong Kong Police Force (RHKP) have developed an intelligence data gathering and collating system. Built on top of this database is an enquiry system, which uses normal 80 by 24 screens. Because the data volumes are very large, it is quite difficult for analysts to "see the wood for the trees". The solution chosen to solve this problem, was to build a graphical front-end to the database.

The original database design was based on work done by Anacapa Sciences,[1989a] whose methodology involves deriving graphical representations of the intelligence data. The Anacapa methodology is based on the idea of link analysis.

Link analysis is the construction of a picture which shows the relationships between people, places and other objects of interest. A good analogy is to consider that what link analysis tries to construct is an entity relationship diagram. Indeed, all objects, be they people, buildings, companies, etc., and all referred to as entities. The relationships between the entities, such as "brother" or "company director" are represented as links. On a Link Chart, which is the final product of the link analysis process, entities are represented by circles or boxes and the relationships are represented by lines. The choice between circles and boxes is a subjective one, but people are usually represented by circles and organizations by boxes. The choice is to a certain extent dictated by the requirement that on a Link Chart, the relationships should not cross one another. This requirement is quite hard to satisfy in the general case. In fact, it is very easy to construct a case where this requirement is not satisfiable.

Anacapa Sciences produce all their diagrams on Macintoshs, using a product called CNA. This is a quite simple application which exploits the power of the macintosh, but it is geared more to the production of presentation quality Link Charts and as such there is no underlying database.

We have not gone out of our way to find out if there has been any other work done in this area for the simple reason that considering the nature of the subject area, nobody was likely to tell us.

The Design of IAW

INDEPOL has a number of characteristics which either help and hinder the solution to the problem of building IAW. One of the first problems is that INDEPOL thinks that it is talking to a human. By this I mean that it does not have a programmatic interface. This has meant that IAW, which runs on a Sun workstation networked to a mainframe, has to pretend to be a terminal. This has placed a large burden on IAW in terms of "screen cracking". By this I mean the syntactical analysis of 80 by 24 screen representations, which are what we have to deal with.

One of the major advantages of INDEPOL is that it contains what is called the INDEPOL model. This is the meta data which describes the data held in the database. This model is accessible by the user. This means that IAW can interrogate INDEPOL, and find out what data is held on the database, in terms of field names, types, etc. This provides a layer of insulation to IAW, in that it is resilient to changes in the underlying structure of the database.

INDEPOL is very consistent in the way it handles the terminal, to which it thinks it is talking in the case of IAW. This has helped us to reduce the work we have to do in order to interpret the output coming from INDEPOL. The screen is split up into three, namely the input, output and error areas. In addition, error messages are displayed in a consistent fashion. INDEPOL has a query language which IAW uses to make database enquiries which are formatted so as to facilitate their analysis by prolog. All the returning data is delimited by "magic cookies" which enables IAW to cater for both missing and duplicate fields[3]. The returning data is passed through BNF like prolog grammar rules which syntactically analyse the data and extract it for storage within IAW.

A bit of history

We were helped in the design of IAW by Jim May, one of the original designers of INDEPOL. We had imagined that he would help us with interfacing to the database. He ended up doing a lot more. INDEPOL is general purpose because it was designed that way. Great force of mind was brought to bear to make IAW general purpose as well. Having the model accessible made this possible.

At first we saw IAW as a classical C development project. With a generalist influence being exerted, it was decided that IAW should contain an interpreter for a language which would glue the system together. Because we were using coloured pens on a white board at the time, the system was split into "red", which was application independent parts of the system, and "blue", which were the application dependent parts of the system. "Red" code was to be C, and "blue" was to be PostScript. NeWS would have been ideal, but the interpreter was on the wrong side of the client server divide. We looked at various Postscript interpreters, but none met our requirements. Having developed the idea of a "soft" core to IAW, we were loathed to throw it away.

In parallel to all this, another group in the ITC were working on a Prolog system. This had a number of appealing characteristics. It interfaced to C language components. It had a graphics subsystem PCE,[Anjewierden1986a] which although initially used SunView, was in the process of being converted to X and the OPEN LOOK[1989b] "look and feel". We quickly embraced SEPIA,[Meier1988a] as the prolog compiler is called.

IAW is split into modules along these "red", "green" and "blue" divisions. In the "red" section are the communications interface predicates and predicates to perform directory manipulation and other miscellaneous tasks. The "blue" section consists of all the user interface code, which is further subdivided by functionality; the Link Chart, Association Matrix and Indented List all being split into a number of modules each and the various other user interface components going into individual modules.

The "green" code contains all the support machinery for the user interface and also contains the code which holds what we call the "working set". This is the data which defines the state of the system, which not only includes data extracted from INDEPOL but also window object references and the IAW model. This is all controlled by one module, which helped recently when we were able to completely change the data representation with no noticeable changes to the rest of IAW[4].

SEPIA

SEPIA is a prolog system, that was developed at the European Computer Industry Research Centre (ECRC), in Munich, Germany. The ECRC is a joint venture between ICL, Siemens and Bull; the results of the research being available to the three companies to do with as they see fit. ICL has taken SEPIA and is productizing it; that is, doing to it all the things that research programmers hate such as providing documentation and support.

SEPIA is seen by its authors as a next generation prolog system. It has been designed to support the low-level integration of extensions to the standard prolog language. Extensions that were of interest to us in IAW, were the module system, the debugger[5] and the graphics subsystem[6], PCE.

[3] INDEPOL is really a relational database with a number of constraints relaxed, the most important one being that duplicate fields are allowed.

[4] Not entirely true as memory loss by SEPIA was reduced. The version of SEPIA that we are using does not have a garbage collector.

[5] The debugger that is supplied with the current version of SEPIA will not work with interrupt driven code. This rules out about 98% of IAW. Fortunately we have just received the next release of SEPIA which gets around this restriction. Up to now we have been debugging with print statements.

[6] PCE has now been bundled with a window based interface to SEPIA and called KEGI.

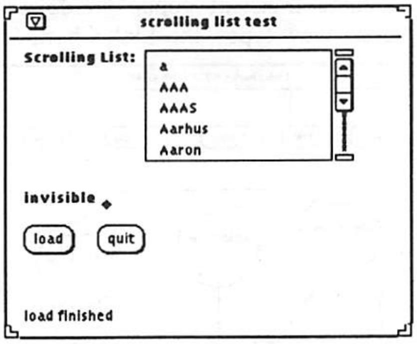
PCE

PCE is very easy to learn because essentially there are only three primitives, new, send and get. The following is a small example which illustrates how easy it is to write graphical applications with SEPIA and PCE.

```
:- global selected_callback/2,
    quit_callback/2, load_callback/2.

top:-
  open_pce,
  new(@frame, frame('')),
  new(@dialog, dialog('scrolling list test')),
  send(@frame, show_footer, true),
  new(@list, scrollinglist('Scrolling List:',
    true,
    cascade(@dialog, selected_callback, 0))),
  new(@text, text_item(invisible, '', none)),
  send(@text, hide, on),
  new(@load, button('load',
    cascade(@dialog, load_callback, 0))),
  new(@quit, button('quit',
    cascade(@dialog, quit_callback, 0))),
  send(@dialog, append, @list),
  send(@text, below, @list),
  send(@load, below, @text),
  send(@quit, right, @load),
  send(@frame, append, @dialog/window),
  send(@dialog, open, point(100, 100)).
```

This small piece of prolog code creates a window which looks thus:



As you can see this create a window which contains a number of objects, a scrolling list containing the beginning of /usr/dict/words, a text field which does not echo back its input and two buttons. The important thing to note about this example is the cascade arguments to the new/2 predicates. These specify what should happen when the user interacts with the object. In this example the appropriate callback routine is called with two arguments. The first argument specifies the object, and the second specifies the user action. In all cases we have delegated the callbacks to the dialog box which contain the objects.

The rest of the code, which follows, is concerned with processing these callbacks. The first section gets the selection from the scrolling list and sends it to the footer.

```
selected_callback(@dialog, Message):-
  get_exclusive_list_selection(list,Selection).

get_exclusive_list_selection(List,Selection):-
  get(@List, selections, @Chain),
  get(@Chain, list, [Selection]),
  send(@dialog, left_footer, Selection).
```

The next section handles loading the scrolling list. load_callback/2 is called when the user presses the load button.

```
load_callback(@dialog, load):-
  send(@dialog, left_footer,'load called...'),
  load_list_from_file(list, "services"),
  send(@dialog, left_footer,'load finished').

load_list_from_file(List, File):-
  (exists(File) ->
    open(File, read, List_file),
    load_list_file(List, List_file),
    close(List_file)
  ;
    true
  ).

load_list_file(List, List_file):-
  repeat,
  read(Line, List_file),
  process_list_line(List, Line),
  Line == end_of_file.

process_list_line(List, end_of_file).

process_list_line(List, Line):-
  send(@List, append_to_sl, Line),
  !.
```

The final section handles the quit button. An OPEN LOOK notice is popped up and the user is given the option to Quit or Cancel. Depending on the answer, the appropriate action is taken.

```
quit_callback(@dialog, quit):-
  quit(Answer),
  !,
  Answer = 'Quit',
  close_pce,
  halt.

quit(Answer):-
  get(@dialog, window, @Window),
  new(@Notice,notice('Do you wish to quit?')),
  send(@Notice, clear),
  send(@Notice, append, 'Quit'),
  send(@Notice, append, 'Cancel'),
  send(@Notice, window, @Window),
  get(@Notice, display, Answer).
```

This small example gives some idea of the power of PCE. If prolog is not for you, then in addition to the prolog interface, there are also interfaces to Lisp and C.

Implementation

IAW is split into three layers, the "red" layer, which is all that portion of the system that is application specific; the "blue" layer, which is all the low level C code to interface to the network and hence the database; and a "green" layer, which is application independent Prolog code.

SEPIA provides a module system, very much along the lines of Modula-2. This we have used to enforce the divisions between the various components of the layers. Only predicates that are explicitly exported from a module can be imported into another module. For example, if we have one file foo.pl which defines a predicate foo/1:

```
:- module(foo).

:- export foo/1.

foo(X):-
    write('The value of foo is'),
    writeln(X).
```

then in order to use it in file bar.pl, we have to import it:

```
:- module(bar).

:- import foo/1 from foo.

bar:-
    foo(10).
```

One problem with this in the current version of SEPIA is that a predicate has to be exported from one module before it can be imported from another. This can cause some file ordering problems when it comes to compiling IAW.

IAW currently consists of 91 separate modules which total 30,000 lines of Prolog and 5,000 lines of C code.

Performance

The machine on which the production system will run, is a 16Mbyte SparcStation. On this machine IAW performs adequately. At the time of writing this paper, there are some acknowledged weak areas in the system, but at the moment our main concern is to get it running reliably. When this has been achieved, we will spend some time trying to speed it up.

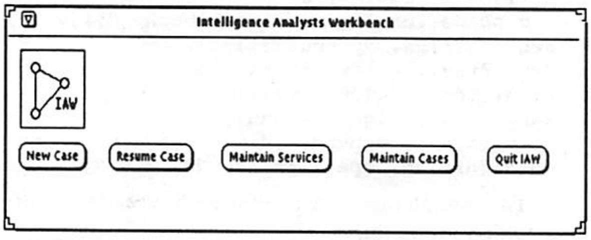
The INDEPOL system that IAW talks to is at the other end of an X.25 network from our site, so performance is not something that we can even consider measuring; with so much of the time waiting for responses to enquiries. We have run IAW on a SparcStation which is on the same ethernet as the INDEPOL service machine and the results have been encouraging.

IAW is going to be introduced into an environment where a Link Chart takes up to two weeks to produce manually. Nevertheless, we are aware that once users get used to the idea of fast response, IAW will have the increase in performance to meet people increased expectations. We will have to deal with this situation when we get to it.

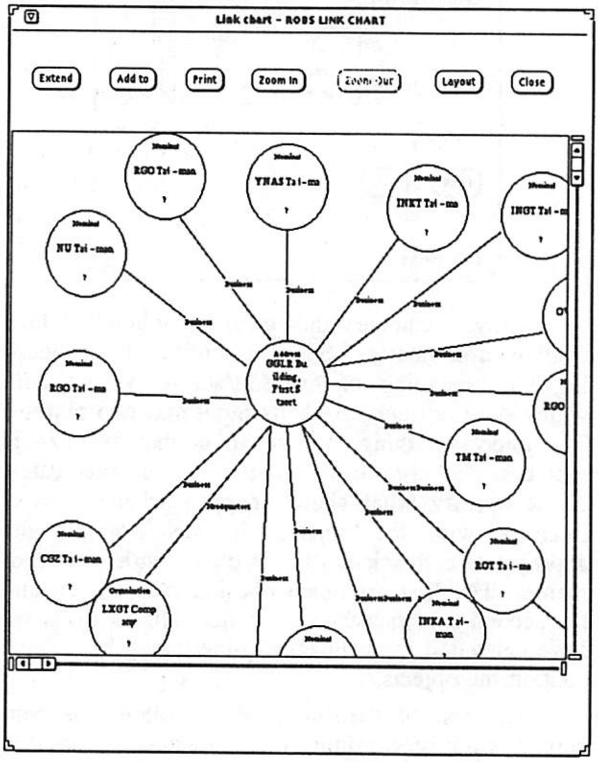
User Interface

The choice of OPEN LOOK was the correct one in my opinion. When we started the project there were a number of choices. My previous work had been on a Cartographical Map editor. For this we used NeWS.[1987a] This was the right choice at the time that we made it nearly three years ago. SunView was another possible choice as this was the default window system on Sun workstations. X was the eventual choice for what I think are obvious reasons. Considering the possible lifetime of IAW, the use of SunView or indeed NeWS would have been very short sighted. Given that X was the window system of choice, this really meant either OPEN LOOK or Motif. This battle is certainly raging as I write this paper, not least within ICL. The decision was made for altogether pragmatic reasons; we had access to an OPEN LOOK implementation, firstly in the form of OpenWindows from Sun and latterly XView, also from Sun, but on the X11 Release 4 distribution.

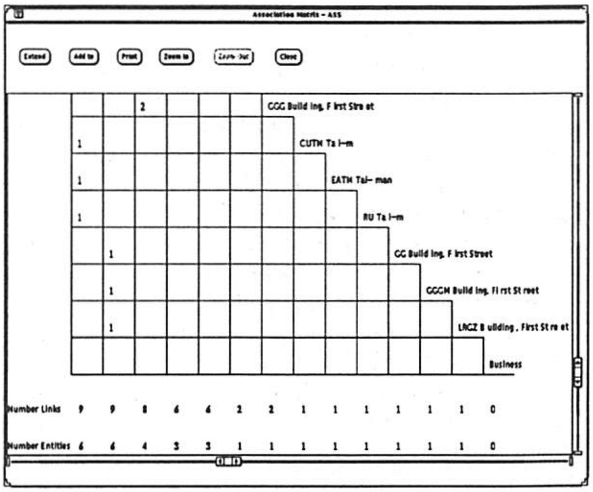
Visually IAW is quite a simple system, consisting of a main window



which via some transient popup windows leads to the three main view types, the Link Chart,

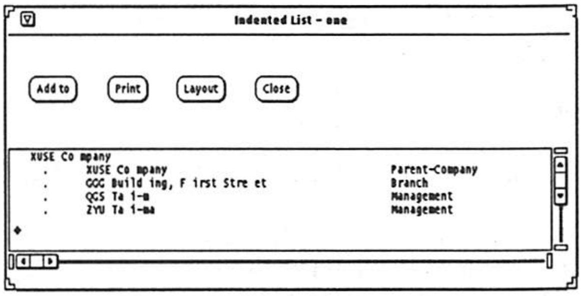


the Association Matrix



			2				CGC Build ing, F irst Stre et							
	1							CUTH Ta l-m						
	1								EATH Tal- man					
	1									RU Ta l-m				
		1									CG Build ing, F irst Street			
		1										CGCM Build ing, Fi rst St reet		
		1											LRCZ B uilding , First St re et	
														Business
Number Links	9	9	8	6	6	2	2	1	1	1	1	1	1	0
Number Entities	6	6	4	3	3	1	1	1	1	1	1	1	1	0

and the Indented List.



```
XUSE Co mpany
   .     XUSE Co mpany                         Parent-Company
   .     GCG Build ing, F irst Stre et         Branch
   .     QCS Ta 1-m                            Management
   .     ZYU Ta 1-ma                           Management
◆
```

Every interaction with the user is done with a PCE dialog box, which is "close" to being an OPEN LOOK command window. This has led to the criticism that there are too many window involved in an IAW interaction. This is something that we are going to have to look at over the next few months.

By utilizing SEPIA's ability to load prolog code incrementally, we will be able to provide the customer with the ability to tailor the user-interface portions of the system.

Conclusions

Building general system has been an education. It requires a lot more thought than we had anticipated. The design phase of IAW was a lot longer than we had planned, but the consequence of this is a robust and tailorable system. When the analysts involved told us that "we knew that they meant" even when they themselves didn't, meant that we had to aim for a very broad target.

The choice of Prolog as an implementation language has been a big success. When the project started, none of us knew any Prolog; apart from small pieces of knowledge, picked up at University. The only problem is that you have to turn your brain inside out to be able to get into the frame of mind necessary to design and write Prolog code. The Open University Prolog Course was invaluable in this regard.[Eisenstadt1988a] I'm sure we are not brilliant prolog programmers, but we are competent.

OPEN LOOK, and the style Guide in particular,[1989c] removed from us a lot of the problems associated with the design of the user-interface.

Acknowledgements

IAW has been a team effort from the very start. Firstly I would like to thank Jim May, of ICL Future System, who has taught me more than a little about how computer systems should be built. Next I would like to thank the IAW team, Robert O'Dea, Mark Stephens, Catherine Tierney, Anne Greally and Dianne Fitzpatrick. Next I would like to thank by boss, Aidan Daly, who puts up with much. Finally I would like to thank my wife Emer, who puts up with even more. I won't be home too late tonight.

This paper is for my son Peter, December 16-19, 1989.

References

1986a. *INDEPOL Reference Manual.* 1986.

1987a. *NeWS 1.1 Manual.* 1987.

1989a. *Advanced Computer-Aided Intelligence Analysis.* 1989.

1989b. *AT&T OPEN LOOK Graphical User Interface Specification Guide Release 1.0.* 1989.

1989c. *AT&T OPEN LOOK Graphical User Interface Style Guide Release 1.0.* 1989.

Anjewierden1986a. A. Anjewierden, "An Overview of PCE-Prolog," University of Amsterdam Technical Report (1986).

Eisenstadt1988a. M. Eisenstadt, *Intensive Prolog.* 1988.

Meier1988a. M. Meier, A. Herold, and D. Henry de Villeneuve, "SEPIA - An Extendible Prolog System," ECRC Technical report (1988).

Simon Kenyon has been a UNIX and graphics consultant with ICL for the last three years. Prior to that he held a number of positions with PRIME, the National Software Centre (an Irish state body helping the software industry) and Ferrani Computer Graphics. He has a BA(Mod) in Mathematics from Trinity College, Dublin. He has been married to Emer for 8 years and has two children, Timmy aged five and Rob aged three. Reach him electronically at simon@icl.ie .

USENIX Summer Conference
June 11-15, 1990
Anaheim, California

Evolution of the U-code Compiler Intermediate Language at Hewlett-Packard

Paul Chan, Manoj Dadoo, Vatsa Santhanam
– Hewlett Packard

ABSTRACT

U-code is a simple stack-based compiler intermediate language developed several years ago for use in portable Pascal and FORTRAN compilers. Several companies including Hewlett-Packard have used U-code to implement production compilers. Through the years, the original U-code definition has been extended at Hewlett-Packard to support new languages and compiler functionality.

Recently, Hewlett-Packard proposed a variant of U-code for use as an Architecture-Neutral Distribution Format (ANDF) in response to Open Software Foundation's Request for Technology.

This paper provides a retrospective look at the evolution of U-code at Hewlett-Packard and discusses its suitability as an ANDF.

Introduction

U-code, short for Universal Pascal Code, is a simple stack-based compiler intermediate language developed in the late seventies for use in portable Pascal and FORTRAN compilers. After verifying the lexical, syntactic, and semantic correctness of the source program, compiler front-ends generate U-code instructions. The compiler back-end maps U-code instructions into instruction codes for the target machine.

By generating U-code, a compiler front-end achieves a degree of machine independence. Front-ends can be retargeted to generate U-code for different machines by changing a relatively small fraction of the compiler source. This makes the front-end reusable for different machines, saving development cost and time.

U-code provides compiler front-ends with a simple memory model, a variety of data types, a virtual expression stack for performing arithmetic and logical operations, and instructions for altering the flow of control.

U-code's simplicity, extensibility, and retargetability have made it an attractive compiler intermediate language. In particular, these advantages have lured computer system developers such as Peritus, MIPS, and Hewlett-Packard to implement U-code-based production compilers.

In the early eighties, Hewlett-Packard adopted U-code as the common intermediate language to be used by compilers for a host of high-level languages running under both its HP-UX and proprietary operating systems, making several extensions in the process. The extended version of U-code has come to be known as *HPcode* [HPcode 89].

A variant of HPcode has been proposed as an Architecture-Neutral Distribution Format (ANDF) in response to a Request for Technology issued by Open Software Foundation. An ANDF would allow 'shrink-wrapped' distribution of software applications to different hardware platforms. The availability of shrink-wrapped software, a primary reason for the success of personal computers, could help UNIX-based systems penetrate the mass consumer market. The feasibility of an HPcode-derived ANDF needs to be established. However, the simplicity and proven extensibility of the basic U-code model leave it well-positioned for success.

This paper provides a retrospective look at the evolution of the U-code compiler intermediate language at Hewlett-Packard. The paper consists of five major sections. Section 1 presents a brief overview of U-code as it was originally defined. Section 2 chronicles the evolution of U-code and cites specific uses of the intermediate language in Hewlett-Packard compilers. Section 3 describes various general and language-specific HPcode extensions. Section 4 contains some implementation notes based on our experience in developing HPcode-based compilers for the PA-RISC architecture [Lee 89]. Section 5 describes how HPcode can be adapted to serve as an ANDF.

1. Overview of U-code

U-code can be regarded as an assembly language for the *U-machine* - a hypothetical stack machine. The U-machine provides the following memory abstraction:

1. A read-only memory area for storing instructions and constants.
2. An expression stack for carrying out most computations.
3. A static storage area for global variables and local variables that retain their values from one invocation of a procedure to the next.
4. A memory stack for pushing and popping stack frames (activation records) on procedure calls and returns. Stack frames are used for storing local variables, parameters, and compiler-generated temporaries.
5. A memory heap for dynamic allocation of data objects.

The U-machine also provides instructions that can be categorized as follows:

- instructions that move data between memory and the expression stack
- instructions that operate on the expression stack (e.g. arithmetic, logical, and set operations)
- jump instructions that alter the flow of control
- call instructions that alter the flow of control as well as the data environment
- miscellaneous instructions that provide static information about procedures, as well as back-end directives and options

The term 'U-code' is used in the rest of this paper to collectively refer to the memory abstraction and instructions supported by the U-machine.

Expression Stack

The expression stack is the central data structure operated on by U-code instructions. Most U-code instructions receive their operands from this stack and push results onto this stack. Several instructions are also available for moving data between the expression stack and the other memory areas. Additionally, U-code memory addresses, labels, procedure descriptors, constants and constant addresses can be loaded onto the expression stack.

An expression stack may not necessarily exist on the target machine. On register machines, the expression stack may be modeled in registers. Care must be taken by back-ends, however, to preserve the semantics of the expression stack. Values loaded onto the expression stack are treated as 'copy' operations. Once a value is copied onto the expression stack, it is not modified by memory operations.

An important restriction applies to the use of the expression stack. Specifically, branching instructions and instructions defining U-code labels require the expression stack to be empty. This restriction helps reduce back-end implementation complexity. In particular, the back-end does not have to determine all possible jump sources and expression stack states for each label.

Note that procedure calls may occur with a non-empty expression stack. However, the elements on the expression stack at the start of the procedure call sequence can be accessed only after returning from the procedure call.

U-code Data Types

Data objects in U-code can be of several types. Historically, U-code data types have been represented by alphabetic characters. Examples of U-code data types include type 'I' for integers, 'L' for unsigned integers, 'R' for reals, 'M' for records or arrays, and 'S' for sets.

For a given U-code implementation, the sizes of objects on the expression stack are uniquely determined by their data types, except for data types 'M' and 'S'. Although data types 'M' and 'S' may be specified as any length, each implementation defines characteristics of the length which will allow more efficient code generation.

For all arithmetic instructions, the data types of the operands must match. Compiler front-ends are responsible for emitting U-code data type conversion instructions where necessary.

Addressing Memory

U-code instructions access data objects in memory by specifying U-code addresses. A U-code address consists of four elements:

<memory type, block number, offset, length>

The memory type identifies which memory area is being accessed and is generally represented by an alphabetic character, much like U-code data types. Common U-code memory types include 'S' designating the Static storage area, 'M' for the Memory stack area, 'P' for the formal Parameter area and 'R' for the fast memory (Register) area. Some U-code implementations define additional memory types such as 'F' for a medium-fast memory area and 'C' for the constant memory area.

The 'block number' component of a U-code memory address is a numeric identifier. Each memory area is sub-divided into regions or blocks that are assigned block numbers by the front-end. Typically, a block number is assigned for the 'M' area of each procedure as well as the formal parameter 'P' area. All Pascal global variables usually reside in a single block in the 'S' area. In FORTRAN, a block number is typically assigned for each COMMON area.

U-code supports the scoping rules found in block-structured languages like Pascal and Algol. Within a procedure, it is possible to access formal parameters and local variables of lexically enclosing procedures by specifying the appropriate block number. It is the U-code back-end's responsibility to make use of the underlying display mechanism in translating such non-local accesses. Architectural details of this nature are not visible at the U-code level.

The offset and length components of a U-code address specify the location and size of the data object being referenced within the memory block. The units used for the offset and length can be expressed in terms of words, bytes, or bits.

U-code Program Structure

A U-code program consists of a collection of U-code procedures. The order of procedures in a U-code program is not significant. In particular, implicit procedure nesting is not allowed in U-code. Instead, the lexical level and block number of each procedure and its enclosing scopes are explicitly specified at the entry point.

U-code Instructions

U-code provides compiler front-ends with a rich instruction set. The major instruction categories are outlined below. A more complete and precise description of the U-code instruction set can be found in [Nye 81].

Memory Reference Instructions

All U-code instructions that reference memory specify a U-code address as an instruction parameter and most also specify the data type of the object being referenced. Instructions are provided for loading and storing expression stack items either directly or indirectly. Addresses of constants and memory items can be loaded for use in indirect memory references. For indexed loads and stores, an instruction is provided to compute the address of array elements based on the index value. Additional indirect memory reference instructions are provided to perform block moves and arithmetic comparisons of contiguous blocks of memory.

Jumps

Front-ends can define U-code labels within a procedure body. U-code labels can be the target of both local and interprocedural jumps. Interprocedural jumps are allowed only to labels defined in procedures in an enclosing scope, as per Pascal semantics. Local jumps into the middle of a procedure call sequence are disallowed.

U-code provides simple unconditional and conditional jump instructions. Additionally, to help implement Pascal *case* statements using jump tables, U-code provides an indexed jump instruction into a table of unconditional jumps.

Procedure Calls

In U-code, procedure calls are performed using a standard instruction sequence. The call sequence is initiated by an MST (mark stack) instruction, and optionally followed by code for actual parameter evaluation. The call sequence is normally terminated by a CUP (call user procedure) instruction that transfers control to the called procedure. Function result values are returned on the expression stack. It is possible to perform nested function calls using nested MST-CUP instruction sequences. Indirect procedure calls are also supported.

At the entry point and exit point of a routine, PSTR (pseudo-store) and PLOD (pseudo-load) instructions are used to associate parameters and return values on the expression stack with the assigned memory locations. The RET (return) instruction returns control back to the caller.

Expression Evaluation

U-code provides a variety of instructions to perform arithmetic, logical, and set operations on the expression stack. Unary operand instructions include boolean inversion, arithmetic negation, square, absolute value, rounding, set size modification, bounds check, data type conversion, as well as increment and decrement by constants.

Instructions that pop two elements off the expression stack and push a result include those that perform the usual arithmetic and logical operations as well as set intersection, set union and set inclusion. Instructions that evaluate the larger or smaller of two values are also provided for convenience.

U-code also provides instructions to alter the expression stack state. It is possible to delete, duplicate, or swap items on top of the expression stack.

Miscellaneous Instructions

U-code defines several other 'non-executable' instructions that provide static information about procedures, as well as back-end directives and options.

Examples of such instructions include specifiers for the local frame size and the lexical level of procedures, import and export requests for variables and procedures, static initializers, and code section delimiters.

Implementation-specific options such as optimization level, assembly output, or symbolic debug support can be passed by the front-end to the back-end through an option specifier. Instructions for correlating machine code offsets with source lines and passing symbol table information to the back-end are also available for symbolic debug support.

U-code provides an UNK (unknown) instruction for implementation-specific extensions to the base language. Some of the early U-code implementations also made use of instructions (e.g. MUSE, MSET) to communicate memory aliasing information for certain U-code instructions (such as procedure calls). This mechanism was used primarily to facilitate optimization.

A Simple Example

The simple example shown in Figure 1 and the annotated U-code translation shown in Figure 2 illustrate the general flavor of a U-code representation of high-level source code – in this case, Pascal. For readability, U-code memory addresses are represented by symbolic names enclosed within angle brackets. In-line explanatory comments are introduced by a semi-colon after the U-code instruction.

```
program test;
var
    a,b : integer;
function sum (op1,op2:integer):integer;
  begin
        sum := op1 + op2;
  end;
begin
    b := 5;
    a := b + sum(3,b);
end.
```

Figure 1: Pascal Source

2. Historical Review

U-code was derived from P-code, an intermediate language proposed by Wirth for use in a Pascal compiler [Wirth 71][Jensen 73][Nori 75]. P-code, like U-code, provides instructions that operate on a hypothetical stack machine.

In the late seventies, a group of researchers at UCSD developed U-code by modifying P-code to facilitate optimization [Perkins 79]. The early uses of U-code include an experimental back-end developed by UCSD for a Cray 1 machine and a back-end for the S-1 machine developed at Stanford [Nye 81]. The first languages supported were Pascal and FORTRAN [Chow 80].

In Hewlett-Packard, U-code was first adopted by the Universal Compiler System project in early 1981 and later extended into HPcode. Development of HPcode-based compilers started in 1982. By 1983, an HPcode-based Pascal cross-compiler was available for the HP 150 computer product (an Intel 8086 implementation). In January 1985, Hewlett-Packard released its first HPcode-based production compiler for HP FORTRAN 77 on the HP3000 stack machine. A few months later, an extended-BASIC compiler was released.

Between 1985 and 1986, Hewlett-Packard implemented almost all of its production compilers on the PA-RISC architecture using HPcode. Languages supported include COBOL, FORTRAN, Pascal, RPG, BASIC, SPL2 and PSPL (proprietary system languages), and TRANSACT (a 4th generation language). In 1989, HP implemented an HPcode-based Ada compiler. Currently, HPcode-based C and C++ compilers are under development for the PA-RISC and Hewlett-Packard's Motorola 680x0 platforms.

3. HPcode Extensions

HPcode contains both general and language specific extensions to the original U-code definition. These extensions were made in the course of developing production-quality compilers for several languages on four different architectures: HP 3000, PA-RISC, Intel 8086 and the Motorola 680x0 family.

The original U-code definition was found to be generally adequate in implementing most language features. New instructions were defined in cases where existing instructions could not be conveniently combined to implement certain language features. Other extensions were made for achieving better code quality.

General Extensions

The original U-code definition allows high-level 'case' statements to be implemented as jump tables, with explicit unconditional jumps to the code sequence for each distinct 'case' value. For greater efficiency, HPcode allows 'case' statements to be specified as a series of case value *ranges*, each range being associated with a distinct code sequence.

This allows front-ends to translate high-level case statements in a straight-forward manner. At the same time, the back-end has the freedom to analyze the case ranges and choose the best code sequence for the target machine. In particular, the case statement can be implemented as a cascaded-if, an indexed jump table, a jump table indexed using a binary search, or in some other way.

U-code provides instructions for defining actual and formal parameters as well as function return values. These instructions had to specify a U-code address, forcing front-ends to make certain assumptions about the underlying stack frame layout. HPcode provides a more architecturally neutral mechanism for identifying parameters and return values. With the added requirement of specifying parameters in the correct order, front-ends just need to provide size information, allowing the back-end to compute the final stack frame offsets.

This extension enabled the same front-end to be used for both the HP 3000 and PA-RISC architectures, localizing the impact of changes in the underlying software calling convention to the back-end.

Language Specific Extensions

HP Pascal

The original U-code definition provides adequate support for the features of the Pascal language such as sets, nested procedures, and case statements. However, new HPcode instructions were needed to support HP's enhanced Pascal implementation (*HP Pascal*).

The most noteworthy language extension in HP Pascal is the *try* statement, which supports error-recovery. A *try* statement defines a block of code where run-time exceptions cause control to be transferred to an associated *recover* statement. Exceptions occurring outside a *try* statement cause control to be transferred to the nearest dynamically enclosing *recover* statement.

HPcode provides new instructions to mark *try* and *recover* blocks as well as instructions to explicitly transfer control to *recover* statements.

HP FORTRAN 77

U-code provides support for most FORTRAN features. However, HPcode extensions were necessary for implementing 3-way branches, assigned GOTO statements, COMPLEX data types and the MIL-STD-1753 extensions found in the HP FORTRAN 77 implementation.

HPcode provides a new 3-way conditional jump instruction to implement the FORTRAN arithmetic GOTO statement. To implement a 3-way branch using U-code jump instructions, the test value has to be stored before the first compare and re-loaded before the second compare (since the stack needs to be empty on jumps). The new instruction circumvents this situation which could result in inefficient code.

To implement the FORTRAN assigned GOTO statement, HPcode defines an instruction to load the address of a label and an instruction to jump to a target specified by a label address.

Two new instructions were added to HPcode to support the COMPLEX data type. These instructions allow a COMPLEX number to be decomposed and re-synthesized out of a pair of real values. COMPLEX arithmetic is performed on the decomposed pair of real values, sparing all arithmetic instructions from having to support a new data type.

HP FORTRAN 77 provides MIL-STD-1753 extensions for bit manipulation. HPcode defines new instructions for logical, arithmetic, and circular

```
BGN  PROGRAM                                          ; begin compile
sum ENT dtype=I level=2 block=3 params=2              ; entry point of function add
  PSTR I <op1>                                        ; pseudo-store parameter 1
  PSTR I <op2>                                        ; pseudo-store parameter 2
  LOD  I <op1>                                        ; load op1
  LOD  I <op2>                                        ; load op2
  ADD  I                                              ; integer add
  STR  I <sum_rslt>                                   ; store result
  PLOD I <sum_rslt>                                   ; load function result
  RET                                                 ; return
  DEF  M 4                                            ; define 4 bytes of local space
  END  sum                                            ; end of sum body

PROGRAM ENT dtype=P level=1 block=2 params=0          ; entry point of program
  LDC  I 4 5                                          ; load 4-byte integer const 5
  STR  I <b>                                          ; store integer to b
  LOD  I <b>                                          ; load integer b
  MST  2                                              ; prepare to call level 2 func
  LDC  I 4 3                                          ; load 4-byte integer const 3
  PAR  I <sum_param1>                                 ; pass as param 1 of sum
  LOD  I <b>                                          ; load integer b
  PAR  I <sum_param2>                                 ; pass as param 2 of sum
  CUP  I 3 sum 2 1                                    ; call sum (block 3, 2 params, 1 ret value)
  ADD  I                                              ; integer add (b + return value of sum)
  STR  I <a>                                          ; store integer to a
  DEF  S 8                                            ; define 8 bytes of static space
  END  PROGRAM                                        ; end of program body
  STP  PROGRAM                                        ; end compile
```

Figure 2: Annotated U-code Translation

shifts, as well as instructions for extracting, depositing, testing, setting, and clearing specified bits. Some of these instructions proved useful for other languages such as C and PSPL.

COBOL

The original U-code definition does not provide enough support for the COBOL language. HPcode extensions were necessary to support PA-RISC COBOL and RPG compilers.

Load and store instructions in HPcode were extended to allow COBOL data types. HPcode also provides several instructions for decimal operations including validation, comparison, conversion and decimal arithmetic. COBOL paragraphs, which are somewhat similar to open subroutines, are also supported in HPcode.

Ada

HPcode extensions were added to support the concept of separate packages as well as exception handling and parameter passing as defined in Ada.

In Ada, packages are used to implement data abstractions and are included by procedures that make use of the abstraction. However, the package body, consisting of routines and variables used to implement the data abstraction, may be compiled separately.

For better code efficiency, HPcode provides extensions that allow the back-end to allocate all package body objects (including objects of separately compiled packages) in the parent procedure's stack frame. Extensions were also added to support nested procedures that Ada allows to be compiled separately.

A simple extension of the error-recovery support added for HP-Pascal was sufficient to meet most of Ada's exception handling requirements. Additional instructions were provided to check and raise error conditions as defined by Ada as well as to ensure that exceptions occur in the correct region even in the presence of optimization.

ANSI-C and C++

Hewlett-Packard is currently implementing HPcode-based compilers for ANSI-C and C++. With the extensions for Pascal and FORTRAN, HPcode was well-positioned to support C and C++. Some new instructions were added to streamline front-end code generation and improve code quality for certain C expressions.

C and C++ introduce language features that have no counterparts in other languages. One such feature is the post-increment operator. The U-code instruction sequence for a C expression of the form **p++* can get complicated requiring excessive expression stack manipulation. To alleviate this problem, HPcode allows front-ends to define parameterized macros for commonly used instruction sequences. U-code instruction sequences for the '&&', '||' and '? :' operators are also complicated due to the requirement that the expression stack has to be empty on jumps. While macros can hide some of the complexity, new HPcode instructions were defined to improve final code quality. New instructions were also added to support initialization of static variables with addresses.

In summary, U-code has proven to be simple to extend for supporting new languages and architectures.

4. Implementation Notes

As mentioned earlier, Hewlett-Packard adopted U-code in the early 1980s for use in production compilers for the HP3000 architecture. When Hewlett-Packard started developing its first PA-RISC implementations in the early to mid 1980s, the company's language organization was faced with the challenge of quickly delivering compilers for the new machines. U-code's simplicity, elegance and promise of retargetability proved to be a tremendous asset in rapidly porting existing HPcode front-ends to the PA-RISC architecture.

The HPcode compiler implementations are discussed below with emphasis on the PA-RISC back-end.

HPcode Compiler Structure

Like many early U-code implementations, HPcode compilers for the HP3000 stack architecture use a two-process model where the front-end process writes HPcode instructions in binary format to a message file which is read concurrently by the back-end process. This is similar to the use of 'pipes' on UNIX systems.

HPcode compilers for PA-RISC, however, use an integrated model where the front-end and back-end components are linked into a single executable file. The front-end emits HPcode instructions by calling procedures in the back-end. The integrated model speeds up the compilation process by not creating an intermediate file.

While the released compilers use an integrated compilation model, HPcode instructions can optionally be written out to an intermediate file. This file can be subsequently processed by a stand-alone back-end or viewed for debugging purposes. The file-based interface has proven to be very useful in identifying front-end code generation problems and quickly resolving compiler defects.

PA-RISC HPcode Back-end Structure

Figure 3 illustrates the overall structure of the PA-RISC HPcode back-end in the form of a data-flow diagram. Each of the circles shown in Figure 3 represents a back-end compilation phase, while intermediate data structures are depicted within parallel lines.

HPcode vs. Quadruples

The first phase, labeled *Tuple Generator* in Figure 3, translates HPcode instructions generated by the front-ends into a linear list of quadruples. By modeling the effects of each HPcode instruction on a virtual expression stack, implicit operands of the stack-based HPcode instructions are expressed as explicit operands of the quadruples. The quadruples form the basis for further translation into register-based PA-RISC instructions. It is interesting to note that other back-end implementations have also transformed stack-based compiler representations into some other intermediate data structure for the purposes of code generation and optimization [Davidson 84].

Most quadruple operands are pointers to symbol table entries created by the Tuple Generator for constants, compiler temporaries, label definitions, etc. The Tuple Generator also performs some simple optimizations to facilitate low-level code generation.

One common optimization is to *delay* or suppress the generation of certain load quadruples. For instance, suppressing 'load-constant' quadruples facilitates the generation of PA-RISC instructions that use immediate operands (e.g. the add-immediate instruction), and saves extra instructions needed to generate the constant.

Great care is taken to ensure that the HPcode semantics are not violated when generating quadruples with implicit loads. In particular, the HPcode sequence 'LOD <a>, LOD <b>, STR <a>, STR <b>' which swaps the values of <a> and <b> can not be translated into 'store b→a, store a→b'. Another situation that demands care involves nested procedure calls. In the software calling convention used for PA-RISC, procedure parameters are stored in the caller's activation frame. In the example shown below, the first five parameters to *outer_call* must be maintained in temporaries until the call to *inner_call* is completed – otherwise they would be destroyed. Note that HPcode hides such details from compiler

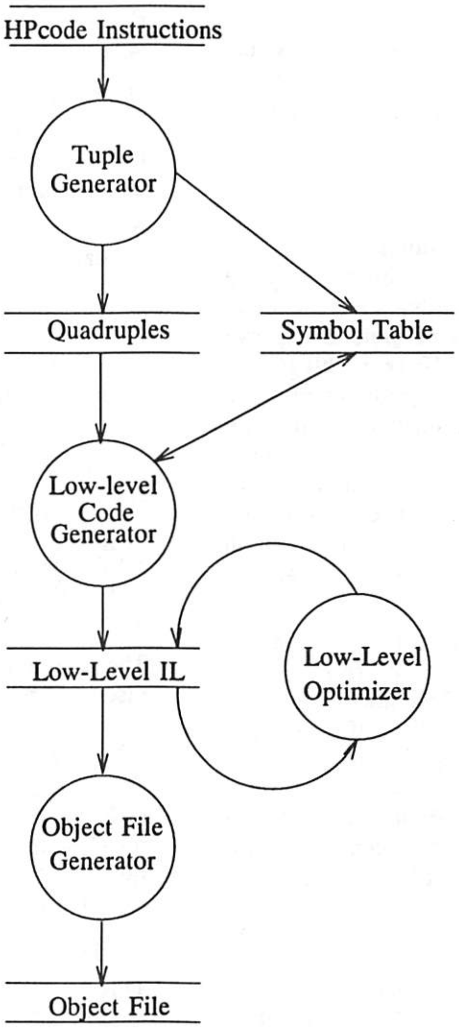


Figure 3: HPcode Back-end Structure

front-ends.

outer_call (1, 2, 3, 4, 5, inner_call(6, 7, 8, 9, 10), 6);

Code Generation

The PA-RISC low-level code generator shown in Figure 3 translates the list of HPcode quadruples into a PA-RISC machine-level representation. Several ad-hoc instruction selection strategies that make use of compile-time information are employed during this translation.

One instance where compile-time information is heavily exploited is the code generation for COBOL decimal operations. Even though PA-RISC provides minimal support for decimal operations, the code generator is able to recognize special cases and generate very efficient code by using compile-time information [Pettis 87].

Incrementing a COBOL unpacked decimal (ASCII display) data type is typically performed by converting to packed decimal, performing a packed decimal addition, and then converting back to an unpacked format. On PA-RISC systems, the low-level code generator recognizes that typically a carry will not be generated by a decimal addition and only the least significant digit of the number will have to be altered. It then becomes possible to generate extremely fast in-line code sequences to perform the operation.

Optimization

Most of the global and local optimizations are performed not at the HPcode level, but on the machine representation emitted by the low-level code generator [Coutant 86b]. It is certainly possible to improve code quality at the U-code level [Perkins 79] [Chow 83]. However, we believe that it is more appropriate to perform optimizations at the machine instruction level. This is true even for seemingly machine-independent optimizations such as common sub-expression elimination. For instance, loads and stores of global variables require two instructions on PA-RISC, one of which can often be shared among multiple references.

The low-level optimizer shown in Figure 3 makes use of language specific aliasing information that describes the side-effects of memory reference instructions [Coutant 86a]. This alias information is communicated from the front-ends to the optimizer through HPcode. HPcode instructions that reference memory are annotated by the front-ends with the appropriate alias information. The mechanism used is very similar to the MSET and MUSE instructions used in early U-code implementations.

HPcode and PCC trees

The semantics of HPcode instructions for the most part allow straight-forward translation into other intermediate representations. Besides translating HPcode into quadruples, translation into PCC (Portable C Compiler [Johnson 79]) tree fragments has also been implemented. The motivation behind this was to provide a quick interface to existing PCC tree-based back-end technology for HP's Motorola 680x0 (MC68K) implementations. This extension allows the same front-end to be used in both PA-RISC and MC68K compilers without having to implement a new HPcode back-end for the MC68K platform.

5. Adopting HPcode as an ANDF

Background

Open System Foundation (OSF) recently issued a request for technology for an 'architecture neutral distribution format' (ANDF) for software distribution [RFT 89]. This was motivated by the vision of identical shrink-wrapped software distribution of an application for all OSF systems regardless of the underlying hardware platform. The availability of shrink-wrapped software, a primary reason for the success of personal computers, could help OSF systems penetrate the mass consumer market.

Figure 4 illustrates the basic ANDF distribution model.

OSF has laid down several requirements for the ANDF. An effective ANDF has to satisfy the following requirements:

- Hardware and architecture independence
- Support for ANSI-C and eventually other standard languages including C++, FORTRAN, Ada, Pascal, COBOL, and BASIC.
- Support for a variety of architectures and hardware implementations
- Protection of proprietary information
- Minimal performance degradation
- Support for symbolic debug

One approach is to adopt a compiler intermediate language as an ANDF. With this approach, the ANDF producer is essentially a compiler front-end and the ANDF installer is essentially a compiler back-end.

The suitability of HPcode for use as an ANDF is discussed below.

Analysis of HPcode as an ANDF

The current HPcode definition does seem to meet most of the requirements. However, a careful investigation uncovered a few problem areas. The relative strengths and weaknesses of HPcode as an ANDF are discussed below.

Hardware and Architecture Independence

This is an area where the current HPcode definition needs additional work. The current HPcode definition is not completely hardware or architecture independent and has the following shortcomings.

Machine dependent storage allocation

Recall that data objects are referenced using HPcode memory addresses specified by front-ends. Front-ends are responsible for storage allocation and need to know the exact length and alignment requirements for specific data types on the target machine. In the ANDF setting, where multiple target machines can have different data sizes and alignment requirements, front-ends can not generate HPcode instructions that will work correctly on all hardware platforms. The problem is particularly severe if the distributed code invokes natively compiled system library routines.

Machine dependent instructions

Some HPcode instructions are machine dependent. For example, UNK instructions are used by front-ends to generate machine-specific instructions. Some OPTN (option) instructions pass implementation-specific information to the back-end.

Machine dependent data representations

In specifying numeric constants, front-ends use data representations such as IEEE floating-point, 2's complement integers, big-endian byte ordering, etc. Incorrect assumptions about the encoding and size of certain data types necessitate an extra layer of conversion and potential loss of accuracy.

Machine dependent header files

The ANSI-C standard header files contain a number of macros and type definitions [ANSI-C 90]. Even though the interface to these macros and type definitions is machine independent, the definitions themselves are not. In particular, if standard header files are expanded before distribution, the distributed program will be rendered machine dependent. No mechanisms exist in the current HPcode definition to defer processing the standard header files until install-time.

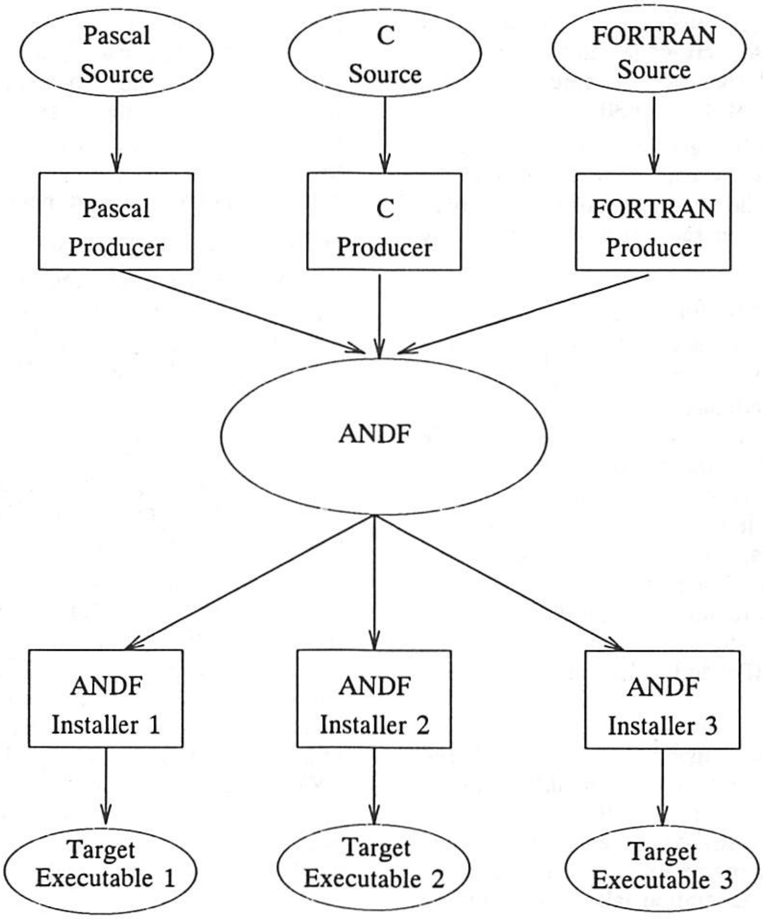


Figure 4: ANDF Model

Support for a variety of languages

HPcode possesses a rich instruction set that supports numerous high level language constructs and data types. Indeed, HPcode supports a wide range of languages including Pascal, FORTRAN, Ada, COBOL, RPG, and BASIC. With the recent extensions to support C and C++, the HPcode definition meets this requirement. Furthermore, HPcode macros facilitate support for new language features requiring lengthy code sequences, without sacrificing compatibility with existing implementations.

However, there are a few constructs that require special attention in an ANDF setting. One good example is type conversion in ANSI-C. ANSI-C has specific conversion rules which depend on the relative data sizes of different data types (e.g. integral promotion). With the current HPcode definition, explicit conversions are required to ensure consistent data types for instruction operands. Front-ends would have to make assumptions about the data sizes in order to perform such conversions, violating the architecture neutrality requirement in the process.

Support for a variety of architectures

Since HPcode is based on a simple stack machine model, it can be easily translated into a variety of machine architectures. HP-code implementations exist for the HP3000 stack architecture, PA-RISC, the Intel 8086, and the Motorola 680x0.

HPcode has proven to be easy to extend. New data types and memory types can be defined and the OPTN instruction can be used to modify the semantics of existing instruction sequences to support new features.

Protection of proprietary information

HPcode is essentially an assembly language for a stack machine. Therefore it provides the same level of protection for proprietary information as binary code. The conversion from source code to HPcode is a one-way mapping. Linear infix expressions are converted into postfix stack expressions. Loop structures are translated into simple conditional and unconditional branches. All variable names except those of global variables are stripped (unless symbolic debug support is required). Consequently, it is practically impossible to reconstruct the original source code from an HPcode translation.

Performance

The compile time performance of HPcode based compilers is comparable to compilers that translate source to machine code directly. Translating HPcode into machine instructions generally accounts for about 20% to 30% of the total compile time. In return, front-end code generation takes less time and is simplified.

In an ANDF setting, it is important that the installation process for low-end target machines not require an inordinate amount of resources in terms of memory usage and time. As such, it is desirable to perform time consuming global optimizations prior to final distribution. This implies that it should be possible to perform machine independent optimizations at the ANDF level.†

The HPcode definition does provide opportunities for machine independent optimization. While Hewlett-Packard compilers do not perform optimizations at the HPcode level, global U-code-to-U-code optimizers have been implemented [Perkins 79] [Chow 83].

As far as run-time performance is concerned, prior experience with HPcode-based compilers indicates that the generated code quality is comparable to that of non-HPcode based compilers. In particular, HPcode allows all the information necessary for optimization to be passed to the back-end.

Symbolic Debug Support

In the current HPcode definition, symbolic debug support is not architected at the HPcode level. HPcode front-ends construct the symbol table in the exact format required by the target debugger and pass the table to the back-end through special instructions. The back-end augments certain fields of the symbol table and writes it out to the object file. This scheme forces the front-end to understand the target debugger requirements. Moreover, the intermediate code containing the symbol table information is not architecture neutral.

HPcode-Plus – A Proposed Solution

To solve all the problems discussed in the previous section, a new version of HPcode called *HPcode-Plus* has been proposed [HP 89]. The major differences between HPcode and HPcode-Plus are outlined below.

Deferred Storage Allocation

A number of the problems mentioned earlier can be solved by deferring storage allocation to the installer (the machine-specific code generator).

To accomplish this, HPcode-Plus provides an enhanced SYM (symbol information) instruction to pass information about types and variables from the ANDF producer to the ANDF installer. This allows variables to be allocated in an architecturally neutral manner. Each SYM instruction specifies type and other information about a symbol. The format of the SYM instruction depends on the symbol type. A symbol can be a dynamic or static variable, a structure or array type, or a field of a structure. Each symbol is assigned a symbol-id. All load and store

†Besides reducing installation time, optimizations done at the ANDF level can help reduce distributed code size. Globally optimized code would also be harder to decipher, and thus provide an added degree of protection.

instructions use this symbol-id to access a variable. Predefined types are provided to specify the data types of variables.

In the HPcode model, an HPcode address and data type are used to refer to data objects. In HPcode-Plus, the address and data type are each replaced by a symbol-id. For example,

```
LOD A M 1 0 32 ; load a 4-byte address from the local area
ILOD I 0 32    ; load a 4-byte integer through the pointer
```

becomes:

```
SYM n local_variable pointer_type int_type
  :
 LOD n
 ILOD
```

where n, pointer_type, and int_type are all integer symbol-ids. Note that this scheme does not require any size or alignment specification. The installer will determine the default data size and alignment based on native data sizes and alignment rules. The result is that the distributed code is architecturally neutral and can invoke native system library routines without problems.

HPcode-Plus frees the producer (i.e. compiler front-end) from requiring machine specific information. The installer can generate symbol tables in the format required by the native debugger using the information provided by the SYM instructions. Any aliasing information required for optimization can also be obtained from the SYM instructions, eliminating the need for special HPcode instructions to pass such information.

Removal Of Machine Dependent Instructions

All machine dependent instructions, including UNK and a number of OPTN instructions, have been either removed from the HPcode-Plus definition or replaced by more general machine independent instructions.

Neutral Data Representation

As mentioned earlier, assumptions about data representation will introduce machine dependencies. In the HPcode-Plus definition, no assumptions are made about internal data representation. In particular, all constants are represented in ASCII format. A compression mechanism can be employed to reduce code size, if necessary.

Deferred Standard Header File Processing

HPcode-Plus provides a macro and constant constructor facility to allow deferred expansion of system macros and type definitions found in the standard header files. Essentially, rather than expanding the system macros and type definitions, the producer special-cases identifiers defined in the header files and assigns unique symbol-ids. When the installer encounters these symbol-ids, it substitutes the local definitions.

ANSI-C Type Conversion

HPcode-Plus provides special instructions to deal with ANSI-C specific conversions rules that depend on the data size of the operands. These instructions handle integral promotion, arithmetic conversion, and type specification for numeric constants.

6. Conclusion

U-code's simplicity, extensibility, and retargetability have made it a successful compiler intermediate language of historical significance. Several companies have adopted U-code for use in production compilers. In particular, at Hewlett-Packard, U-code has evolved from a simple intermediate language for Pascal and FORTRAN into one that can support diverse high-level language features found in COBOL, Ada, ANSI-C, and several other programming languages.

U-code's inherent strengths leave it poised for application to the software distribution problem currently faced by independent software vendors in the UNIX arena. HPcode-Plus, the U-code derivative that has been proposed as an ANDF, retains the strengths of U-code, while either eschewing its machine dependent characteristics or reformulating them in an architecturally neutral context.

If proven to be a viable ANDF, U-code, in its incarnation as HPcode-Plus, will undoubtedly continue to evolve in many ways. Extensions to support vectorization and different levels of parallelization are a definite possibility. The demands placed by ANDF-related tools (e.g. CASE tools) could also shape the intermediate language's future growth.

In summary, U-code has evolved to support multiple languages and architectures in production compilers. With considered modifications, it appears eminently qualified for use as an ANDF and could become an intermediate language of even greater importance.

Acknowledgements

We would like to acknowledge the efforts of the many engineers at Hewlett-Packard's California Language Laboratory (CLL) who were responsible for designing and implementing HPcode extensions over the last several years. Karl Pettis deserves special recognition for his contributions to the definition of HPcode-Plus. We would also like to thank the individuals who reviewed earlier drafts of this paper and the management staff at CLL for their support.

References

[Wirth 71] Wirth, N., "The Design of the PASCAL Compiler," *Software-Practice and Experience* 1:4, pages 309-333, 1971.

[Jensen 73] Jensen, K., and Wirth, N., "An Assembler/Interpreter for a Hypothetical Stack Computer," program listing, 1973.

[Nori 75] Nori, K. V., Ammann, U., Jensen, K., and H. Nageli, "The Pascal (P) Compiler Implementation notes," Institut fur Informatik, Eidgenossische Technische Hochschule, Zurich, 1975.

[Perkins 79] Daniel R. Perkins and Richard L. Sites, "Machine-independent Pascal Code Optimization," EECS Dept., University of California, San Diego, *ACM SIGPLAN Notices*, August 1979.

[Johnson 79] S. C. Johnson, "A Tour Through the Portable C Compiler," AT&T Bell Laboratories, Murray Hill, N.J., 1979.

[Chow 80] F. Chow, P. Nye and G. Wiederhold, "UFORT: A Fortran to U-Code Translator," Computer Systems Lab Technical Report 168, Stanford University, January 1980.

[Nye 81] Nye P., "S-1 U-Code: An Intermediate Language for Pascal and FORTRAN," S-1 Project Document PAIL-8, Computer System Lab, Stanford University, October 1981.

[Chow 83] Frederick C. Chow, "A Portable Machine-Independent Global Optimizer - Design and Measurement," Technical Note No. 83-254 December 1983, Computer Systems Laboratory, Dept. of EECS, Stanford University.

[Davidson 84] J. W. Davidson and C. W. Fraser, "Code Selection through Object Code Optimization," *ACM Transactions on Programming Languages and Systems 6*, October 1984, pages 7-32.

[Coutant 86a] D.S. Coutant, "Retargetable High-Level Alias Analysis," *Conference Record of the 13th ACM Symposium on Principles of Programming Languages*, January 1986.

[Coutant 86b] Deborah S. Coutant, Carol L. Hammond, and Jon W. Kelley, "Compilers for the New Generation of Hewlett-Packard Computers," *Hewlett-Packard Journal*, Vol. 37, no. 1, August 1986, pages 4-18.

[Pettis 87] Karl W. Pettis and William B. Buzbee, "Hewlett-Packard Precision Architecture Compiler Performance," *Hewlett-Packard Journal*, Vol. 38, no. 3, March 1987, pages 29-35.

[Lee 89] Ruby B. Lee, "Precision Architecture," *Computer*, January 1989, pages 78-91.

[HPcode 89] "HPcode Reference Manual," Hewlett-Packard internal document, January, 1989.

[HP 89] "Hewlett-Packard's Compiler Intermediate Language HPcode-Plus - An ANDF Summary Proposal," submitted to Open Software Foundation, June, 1989.

[RFT 89] "Architecture-Neutral Distribution Format: Guidelines for Technology Submissions" - supplement to Request for Technology, Open Software Foundation, August 1989.

[ANSI-C 90] "American National Standard for Information Systems – Programming Language C," Document X3J11/88-159, The American National Standards Institute, February 1990.

Paul Chan received his BS in Social Sciences from Hong Kong University in 1978 and his MS in Computer Science from the University of Hawaii at Manoa in 1981. Since he joined HP he has worked on a number of compiler related projects including design and development of symbolic debuggers, FORTRAN compilers, HPcode code generator, and code optimization. Recently, he has coordinated the Hewlett-Packard's response to OSF's request for technology on ANDF (Architecture Neutral Distribution Format).

Manoj Dadoo received his BS in Computer Science from Indian Institute of Technology at Bombay, India in 1983 and his MS in Computer Science from Rensselaer Polytechnic Institute in 1984. Since then, he has worked in the Hewlett-Packard California Language Lab, where he was primarily involved in the design and implementation of HPcode-based code generator and optimizer for PA-RISC architecture. Most recently, he coordinated the C++ compiler efforts.

Vatsa Santhanam has been a member of the technical staff at Hewlett-Packard since 1984, where he has developed system software for a VLSI digital IC tester and has also worked on the PA-RISC optimizer. He recently investigated interprocedural register allocation techniques and co-authored a paper on the subject. Vatsa is currently involved with Hewlett-Packard's ANDF proposal. He received a Bachelor of Technology degree in Electrical Engineering from the Indian Institute of Technology at Madras, India in 1982 and a Master of Science degree in Computer Science from the University of Wisconsin – Madison in 1984.

USENIX Summer Conference
June 11-15, 1990
Anaheim, California

The Evolution of Dbx

Mark A. Linton – Stanford University

ABSTRACT

Dbx is the standard source-level debugger on most Unix workstations. Over the past six years Dbx has grown from a debugger for interpreted Pascal programs to a debugger for compiled C, C++, FORTRAN, Pascal, and Modula-2 programs. Dbx also has been retargetted to a variety of architectures, including VAX, Motorola 68000, MIPS, IBM RT-PC, IBM 370, Sun SPARC, and Intel 80386.

This paper describes the evolution of Dbx and examines how the organization of Dbx has enhanced its portability and extensibility. The structure of Dbx is based on a set of abstractions that define what a debugger must do, not on a decomposition by language or machine. These abstractions provide greater flexibility in handling the unexpected problems associated with retargetting a program.

Introduction

Dbx is a source-level debugger that runs on most Unix systems. Compared with earlier Unix debuggers such as Adb [1] and Sdb [4], Dbx contains significant advances in the areas of portability, multilingual support, and ease of use. A port to a new machine architecture takes about one week. Dbx can be used with several programming languages, even in the same application. The Dbx user interface provides a high-level set of commands that can be extended using macros.

Dbx did not start out as a portable, multi-language debugger. The original version of Dbx, which was called Pdx, was the author's Master's project at the University of California at Berkeley [6]. Pdx was written to add debugging support to the Berkeley Pascal system [3]. The implementation was targetted specifically at the Pascal language and the Berkeley Pascal interpreter (''px'').

Despite the limited scope of the original project, Dbx has evolved into a much more general debugger without major changes to the original structure. This evolution has been possible because the implementation is decomposed into logical components that encapsulate debugger functionality.

This paper describes the evolution of Dbx from a single-language debugger for an interpretative Pascal system to a debugger that supports five languages and seven architectures. Following an overview of source-level debugging, the paper presents the major components of the implementation and how they have evolved, relating the experiences porting to Dbx different machines. Future debugging tools should expose these components so that several applications can access them, leading to an ''open'' debugging system that is simple, extensible, and powerful.

Source-Level Debugging

The purpose of a debugger is to help the programmer understand why a program, called the **debuggee,** is producing incorrect results. A debugger can show the debuggee's execution state, including values of variables and a list of currently active functions. The programmer can either view a snapshot of execution state saved in a **dump** file or examine a debuggee process as it is running. On Unix, a debugger normally runs in a separate address space from the debuggee to guarantee that an error in the debuggee cannot affect the debugger's state.

A programmer can set a **breakpoint** to suspend execution of the debuggee process when a specified event occurs. The most common kind of event is execution of code at a specified location in the program. When a breakpoint is reached, the programmer can view the execution state and continue execution to the next breakpoint or to the end of the program. Execution is also suspended when a hardware trap occurs, as would be caused by dividing by zero or dereferencing an invalid pointer.

A *source-level* debugger provides these execution views and controls in terms of the program source. For example, the user can display a variable's value by specifying the variable's name and can set a breakpoint by specifying a line in a source file.

Figure 1 shows a Dbx debugging session. The ''% '' indicates a Unix shell prompt; ''(dbx) '' is the Dbx prompt. The first command the user gives is **run,** which starts the debuggee process. Execution stops when a trap called a ''segmentation fault'' occurs. Dbx determines where the error occurred and lists the offending source line. The user gives the **print** command to display the value of the suspect pointer (''ptr'') and finds that it is nil. The user then gives the **where** command to list the stack of currently active functions, from the most recently called to the first (main).

The user decides to run the program again, this time stopping execution before the error occurs. The user sets a breakpoint at line 10 in the source file with the **stop** command and then runs the program. Execution stops at the breakpoint, and Dbx lists the associated source line. The user prints ''ptr'', and its value seems reasonable. However, ''ptr'' is about to be set to the value of ''badptr'', which is nil. The user then exits Dbx with the **quit** command.

Dbx organization

The design of Dbx was guided by the principles of data abstraction, in which a problem is decomposed into types of objects. For a debugger, the fundamental data types are commands, symbols, the source code, the object code, the run-time stack organization, the machine architecture, and the debuggee process. The Dbx organization reflects this set of types. Figure 2 shows the Dbx modules and how they fit together.

Commands

The command interpreter parses and interprets user input. The goal of the command interface is to provide a simple, high-level set of commands that match what the user wants rather than what the machine or operating system provides. For example, a common debugging technique is to set a breakpoint, print some variables when the program stops, and then continue execution. Dbx supports this technique with the **trace** command. The command **trace** *function* displays a message each time *function* is called and another message each time it returns.

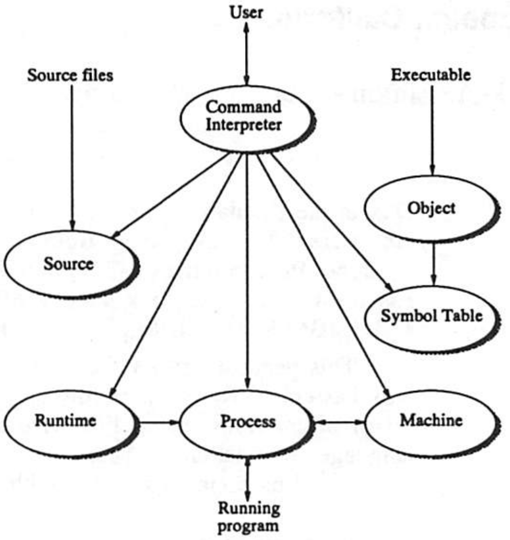


Figure 2: Organization of Dbx modules

Although the command syntax was designed for novice users, a simple macro expansion capability has been sufficient to satisfy experienced users. Unfortunately, the documentation does not emphasize this capability; many users have complained about the long command names, unaware that they can define their own abbreviations.

The implementation of command parsing and interpretation follows the normal structure of a compiler: a scanner reads tokens from input; a parser built with YACC recognizes commands as specified by a grammar; an abstract syntax tree is built during parsing; a pass over the tree checks for correctness; another traversal executes the tree using an evaluation stack.

```
% dbx a.out
dbx version 3.4 of 7/27/87 11:24 (lurch).
Type 'help' for help
(dbx) run
Segmentation fault in P at line 12 in file "source.c"
   12         *ptr = param;
(dbx) print ptr
nil
(dbx) where
P(param = 3), line 12 in file "source.c"
main(argc = 1, argv = 0x7fffe210), line 105 in file "main.c"
(dbx) stop at 10
[1] stop at 10
(dbx) run
[1] stopped in P at line 10 in file "source.c"
   10         ptr = badptr;
(dbx) print ptr
0x7fffe218
(dbx) print badptr
nil
(dbx) quit
```

Figure 1: Sample Dbx session

Symbol lookup is a more complex problem in a debugger than a compiler. The difficulty arises from the need to access variables that are not statically visible. For example, suppose execution is suspended in function *f*, which was called by function *g*. The user may wish to display the values of variables local to *g*. Furthermore, it should be possible to access values in recursive calls.

The Dbx symbol lookup algorithm consists of three steps. First, Dbx looks for the name in a static context, starting with the current function and searching outward as a compiler would. If that search fails, it then looks for the name in a dynamic context, starting with the current stack frame and searching up the call stack. If both searches fail, the symbol could still be defined as local to a compilation unit not in the static context. However, Dbx cannot distinguish between symbols with the same name in several units. Several choices are possible here; it currently picks an arbitrary symbol and prints a warning message.

Dbx also provides a syntax for qualifying names to identify symbols in an outer scope. For example, if execution is suspended in *f*, which was called by *g*, and both *f* and *g* have local variables named *i*, the user can use *g.i* to identify the *i* local to *g*.

It took three attempts at writing the lookup algorithm before all the cases were recognized and handled correctly. The problem was not so much getting the implementation correct as understanding all the possible cases.

Breakpoints and tracing

Most debuggers provide the ability to set a breakpoint that suspends execution when the program reaches a certain location. Such breakpoints support "binary search" debugging, in which one finds an erroneous condition, sets a breakpoint "half-way" between the beginning of execution and the fault point, and then restarts the program. If the execution state looks correct at the half-way point, then the user sets another breakpoint at a three-quarters point and continues execution. The user often checks whether execution is correct by examining the value of one or more variables.

Dbx provides a set of **stop** commands for suspending execution at a breakpoint and a set of **trace** commands for displaying a value during execution. Stop and trace points can be specified at a source line, within a procedure, or when an arbitrary condition becomes true. Trace information can also be requested as each line is executed.

The events module in Dbx implements breakpoints and tracing. The first implementation interpreted stop and trace commands directly. This approach was not flexible enough to handle many cases; in fact, tracing hardly worked at all. The problem was that internal breakpoints had to be created for commands such as **trace** *f*, where *f* is a function. Because the user could interrupt execution at arbitrary locations while debugging, Dbx frequently got confused about what breakpoints should be present.

Event representation

The solution was a major overhaul of event management. The current approach represents all user commands uniformly in terms of events and actions. Dbx translates this representation into low-level breakpoints and interprets associated primitive actions. For example, the representation for the command **trace** *f* could be expressed as

```
when ($proc = f) {
    printcall(f);
    once ($pc = $retaddr) {
        printrtn(f);
    }
}
```

The symbols "$proc", "$pc", and "$retaddr" denote the entry point for a procedure, the value of program counter, and the return address for the current function, respectively. The **when** and **once** keywords distinguish an event of general interest from a one-time-only event, respectively. In this example, the execution sequence consists of the following steps:

1. A breakpoint is set for the entry to *f*.
2. When *f* is called, Dbx suspends execution and displays a message about the call, including parameter values. A second breakpoint is set for the current return address, which is the instruction following the call to *f*. Dbx then continues execution without user intervention.
3. When *f* returns, Dbx suspends execution and displays a message about the return, including the return value. Dbx deletes the return breakpoint and continues execution.

The advantage of this representation is that Dbx can stop execution when a function returns without knowing the address of the return statement(s). Therefore, Dbx can trace functions that do not have debugging information.

Conditional breakpoints

Dbx implements conditional stops and traces correctly, but the algorithm is so slow that using this capability is often not practical. The representation of "stop at 10 if *condition*" is

```
when ($line = 10) {
    if (!$condition) {
        continue;
    }
}
```

The straightforward implementation of this event is to suspend execution at line 10, check the condition from within Dbx, and continue unless the condition is true. Unfortunately, this can create an execution cycle with frequent context switches between the debuggee and Dbx, thereby slowing execution significantly. A far better approach would be to patch the debuggee so that it checks the condition, thus eliminating the context switches.

Conditional tracing in Dbx is often very slow and therefore rarely used. Dbx users have not voiced a strong desire for this situation to be improved, perhaps because they do not realize what they are missing.

Symbols

Dbx uses a uniform representation for symbols. This format is a union of all the types of symbols in different programming languages. The three aspects of symbol management are the external representation that is generated by compilers, the internal representation, and how language-dependent operations are implemented. The Dbx implementation contains a separate module for each aspect.

External representation

Berkeley Unix stores symbol information in a format called “stabs” (for *s*ymbol *tab*le entrie*s*). Following the object code for the functions and data, the symbol information appears in two parts: an array of symbol records and a string table containing the names of all the symbols. Each symbol record contains the offset of the symbol's name into the string table.

The symbol records contain some fields with type information; however, the language information that can be represented in the fields is very limited. For example, there is no way to represent array subscript bounds. To change the symbol table format would have required modifications to many other system tools, including the assembler, linker, and library archiver. Such a change would also have required modifications to any user programs that rely on the format. To modify as few programs as possible, the choice was made to keep the existing format and rely on an important attribute of symbol names---they can be of arbitrary length. Therefore, compilers that generate information for Dbx append an encoding of type information at the end of each symbol's name. This hybrid name is called a “stabstring”.

For example, the C declaration

```
typedef struct {
    int x;
    char* y;
} a;
```

generates the stabstring

```
a:t13=s8x:1,0,32;y:14=*2,32,32;;
```

Dbx recognizes the “:” as separating the variable name “a” from its type. The character immediately following the “:” indicates what kind of symbol it is. In this example, the “t” means “typedef”. A number is associated with every type to avoid replicating information for types that are referenced more than once. This number is unique within the compilation unit. In the example above, the type number is 13. The “s” specifies a structure definition, which is followed by its size in bytes and a list of fields separated by semicolons. Each field has a name, a “:” separator, a type, bit offset, and bit length.

In the example above, the field *x* is defined by the string

```
x:1,0,32
```

Type number 1 refers to the builtin C type **int.** The value of *x* starts at bit 0 in the record and is 32 bits long. The field *y* is defined by the string

```
y:14=*2,32,32
```

A new type with number 14 is defined to be a pointer to type number 2, which is the builtin C type “char”. The value of *y* starts at bit 32 in the record and is 32 bits long.

The original motivation for encoding information in names was a short-term desire to avoid modifying the assembler, linker, and other tools. It was also quick---it took about one week to modify the C compiler to generate the encodings. In retrospect, this approach has had many long-term advantages. As Dbx evolved to support additional languages, stabstrings have been easy to extend. For Modula-2, for example, it was necessary to add information for modules, opaque types, and nested procedures. Because names are arbitrary length and other tools are not concerned with the contents of a name, this information was added by changing only Dbx and the Modula-2 compiler. Thus, what began as a “quick-and-dirty” solution turned out to be a wise design decision.

Internal representation

The Dbx object file module builds a symbol table from the information stored in the executable image. Dbx represents symbols internally with a structure containing a fixed-length section and a varying-length section for information specific to a symbol. The fixed-length section contains the following fields:

name	identifier
language	pointer to procedure vector
class	variable, function, type, ...
storage	register, stack, heap
type	pointer to another symbol
chain	link to next symbol in a list
level	lexical depth
block	pointer to outer scope
next_sym	hash table chain

The varying-length section contains information such as the stack offset for a local variable, code address for a function, and range bounds for a subrange type. Figure 3 shows the representation for several C declarations.

This representation was based on the symbol table organization used by the Berkeley Pascal translator, Pi. Because type equivalence in Pascal is determined by a type's name, it is necessary to represent the entire type graph. This requirement led to a general symbol representation in Pi, which was easy to adapt to other languages. If Dbx had started with the traditional C symbol representation, which packs type information into a single word, it would have been more difficult to support other languages.

A shortcoming of the symbol representation is that size information should be present for *all* symbols. For one or two languages and compilers, symbol sizes can be computed correctly. However, when Dbx was retargetted to machines requiring word alignment or compilers that used different strategies for choosing the size of an enumerated type, it became clear that the compiler should generate size information for all symbols.

Language-dependent operations

The Dbx **print** command evaluates and displays the value of an expression. The display format can vary depending on the language associated with the symbols in the expression. The syntax of Pascal and Modula-2 sets is different, for example. Dbx also provides the **whatis** command to display the declaration of a symbol using the syntax of the symbol's language.

Dbx uses an array of procedure variables to implement language-dependent operations. Every symbol contains a pointer to a language-specific data structure. This pointer is assigned as the symbol is read from the external representation. The language data structure contains the name of the language and an array of procedure variables. A language-dependent operation is called on a symbol by dereferencing the symbol's pointer, indexing for the desired operation, and calling indirectly through the procedure variable. The reason every symbol contains a language pointer instead of having a global language mode is to make sure an operation is always defined. For example, displaying a Pascal set would make little sense in C language mode.

As Dbx has evolved to support new languages and features, several language-dependent operations have been added. When FORTRAN was added, for example, it was necessary to define multi-dimension array subscripting as language-dependent because Pascal and FORTRAN use row-major and column-major addressing, respectively. Another addition arose from the implementation of the **call** command, which allows the user to execute a procedure in the debuggee program. A language-dependent operation was added that tests whether to pass a parameter implicitly by address, because arrays are passed by address in C and by value in Pascal.

Although the support for multiple languages in Dbx is adequate for the current set of languages, several problems remain to be solved. Inline procedures are only partially-supported: Dbx can recognize where execution has suspended within an inline call and display a ''virtual'' stack frame entry, but there is no support to set a breakpoint inside an inline. Language features such as overloading, generics, and inheritance are not supported at all.

Source-address mappings

When a user wants to stop execution at a line in a source file, Dbx must compute the address in the object code that corresponds to the given line. Conversely, whenever execution stops, Dbx must compute the source file and line number that corresponds to the program counter. These mappings are implemented using two binary-search tables, one for files and one for lines. Each element in the file table contains a file name, object-code address, and an index into the line table. Each element in the line table contains a line number and object-code address. Figure 4 shows how the tables are organized.

The tables are sorted by address, from smallest to largest. To compute the source file and line number for a given address, an algorithm similar to a binary search is performed on each table to find the element with the largest address less than the given address. To compute the address for a given source file and line number, first a linear search is performed to find the file's name in the file table. Then a second linear search is performed on the region of the line table to find the given line number. A binary search cannot be used on the line number because a source line may be associated with several object code addresses. For example, a C for-loop typically has two addresses, one corresponding to the beginning of the loop and one corresponding to the increment at the end of the loop.

Process management

The process module encapsulates the operating system mechanisms for controlling execution of the program being debugged. The module is divided into two layers: a high-level interface used by the command interpreter and a low-level interface to machine-specific data such as the values in registers.

The top layer includes operations to run, continue, and single-step a process, as well as to read and write the process' memory. This layer automatically continues execution after handling an implicit breakpoint associated with a trace command. When a dump is being examined (instead of a suspended process), the top layer implements memory access

```
typedef struct A {
    char id[50];
    struct A *next;
} *B;

int f(x)
B x;
{
    ...
}
```

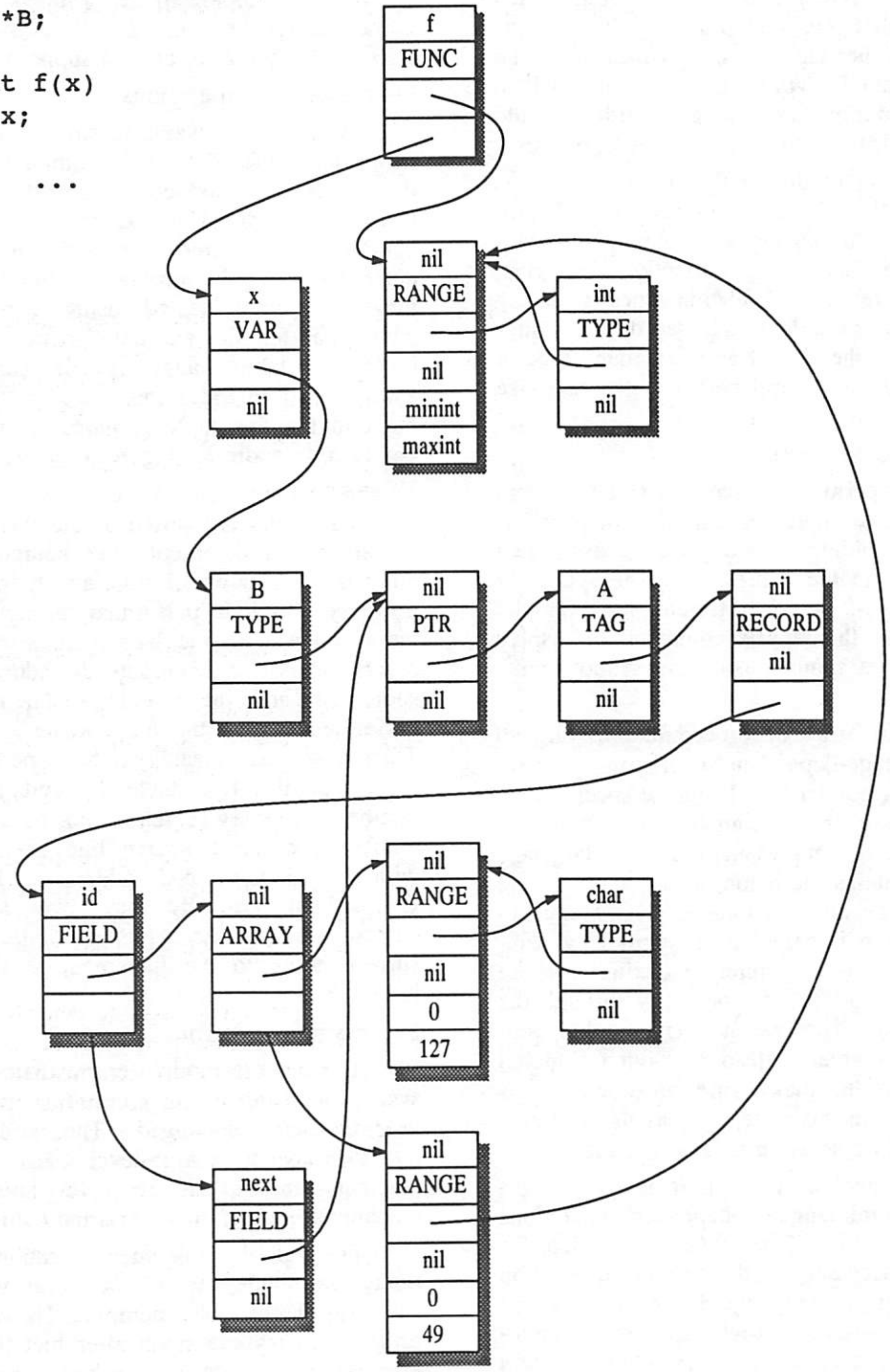


Figure 3: C declarations and symbol representation

with the appropriate file access; otherwise, it passes the request down to the bottom layer.

The bottom layer provides operations to create, destroy, continue, step, read and write memory, and read and write registers. The memory interface can access an arbitrary section of the process' address space. Some versions of Unix support this kind of access directly; other versions allow only single-word accesses.

The bottom layer caches register and memory information to avoid unnecessary accesses to the process' address space. These accesses can be quite expensive on some systems (equivalent to several context switches).

Although the number and kind of registers is machine-dependent, they are accessed through the process layers because of system dependencies. On some Unix systems, for example, it is necessary to use the kernel's per-process data structure to find the register values.

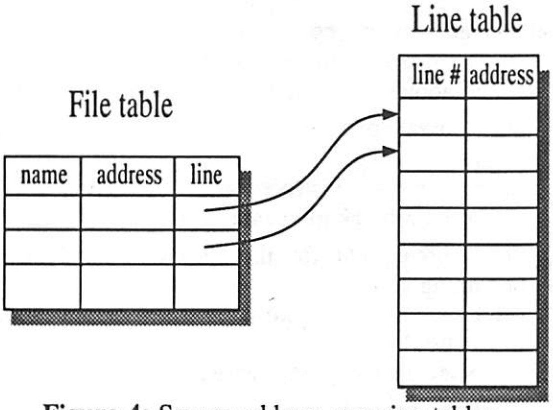


Figure 4: Source-address mapping tables

Run-time information

The major function of the run-time module is to "walk the stack" from the current function through each call back to the start of the program. There are three distinct uses of stack walking. The most obvious use is to list all the active calls when execution is suspended. The second use is to find the stack frame associated with a given function. This search is needed when the user attempts to display a variable local to a function up the call chain. For example, consider the following C code:

```
int g()              int f()
{                    {
  int x = 3;             int y;
  f();                   ...
}                    }
```

If execution is suspended in *f* and the user wants to display *x*, Dbx must find *g*'s activation record on the stack.

The third use of stack walking is for commands that set the frame used to start the search for a function. The **up** command moves the current frame up the stack to the caller of the current function, and the **down** command moves down to the callee of the current function. Originally, Dbx only had a concept of a current function, not a current frame. Consequently, Dbx could only print the values of local variables associated with the most recent call of a recursive function.

Defining a current frame also solves a subtle problem with the **next** command when stepping through recursive functions. Consider the following code:

```
f(x)
int x;
{
        if (x > 0) {
                f(x-1);
                g(x);
        }
}
```

The **next** command single steps across a function call. That is, if a statement is a function call, **next** continues execution up to the statement following the call. The **step** command, in contrast, continues execution to the first statement in the function that is called. In the code above, a naive implementation of a **next** command at the call to *f(x-1)* would continue to the call to *g,* but in the recursive call. To implement the proper semantics for **next,** Dbx continues execution in a loop that terminates when the current frame matches the starting frame.

Stack walking can be more difficult if a function is encountered that does not have compiler-generated information. The difficulty depends on the target architecture and compiler. On a VAX, which has a call instruction that sets up a standard stack frame, it is always easy to walk the stack. On a 68000 or 80386, however, Dbx must decode the instructions at the beginning of the function to determine which registers were saved and to check for optimizations, such as omitting the stack pointer inside a leaf function (a function that does not call other functions).

Some debuggers assume that all functions have debugging information and these debuggers might fail or produce incorrect results when they encounter a function without any information as they walk the stack. In practice, functions are often compiled without symbol information to reduce disk storage or speed up compilation. Therefore, although it is reasonable to give limited information in the presence of such functions, it is unacceptable to terminate or give incorrect information.

Unix signals also complicate stack walking. A signal is an asynchronous event that transfers control to a specified function, called the *signal*handler. When a signal occurs, the operating system stores a special record on the stack before transferring control to the signal handler. Thus, Dbx must detect and skip over this signal record. This logic is not difficult conceptually, but is often time-consuming to implement because the structure of the record is not well-documented on many systems.

Machine interface

The machine module decodes instructions, formats data, and implements breakpoints. No attempt has been made to encapsulate machine dependencies completely in this module. Some machine characteristics, such as the layout of the stack frame, are implemented more easily within the run-time or process modules.

Decoding instructions

Instruction decoding has two uses. First, although Dbx is a source-level debugger, it is occasionally useful to display machine instructions. This feature was originally added to help debug Dbx itself, but it has continued to be valuable.

The second use of decoding is to implement the Next-address function. Given an address *A*, this function determines what value the program counter would have if the instruction at *A* were executed. For branch instructions, Next-address returns the target address; otherwise, it returns *A* plus the size of the instruction at *A*.

Dbx uses the Next-address function to single-step execution. If execution is suspended at *A*, Dbx sets a breakpoint at Next-address(*A*), continues execution (which will stop after executing one instruction), and then removes the breakpoint.

Some machines have a special CPU flag that when set causes a trap after a single instruction is executed. Even when a machine has a special flag, however, using Next-address can eliminate many of the context switches between Dbx and the debuggee. One way to implement single-stepping by source line is to single-step by machine instruction until execution reaches an address that corresponds to the beginning of a source line. Instead of single-stepping, Dbx uses Next-address to compute where execution would be. When Next-address finds a conditional or indirect branch, it continues execution up to the instruction before trying to compute the new address. This approach reduces the number of context switches from potentially slower two per machine instruction to two per branch.

Formatting data

Dbx can read from any address in the process' memory and can display the data in decimal, octal, hexadecimal, character string, or floating point formats. This capability is useful in examining variables that do not have type information.

Formatting data requires knowledge of the machine's bit and byte ordering as well as its floating point representation. The machine module encapsulates this knowledge with a function that extracts a range of bits from a data word. This function is also used to display the value of a C bit field.

Setting breakpoints

To set a breakpoint at location *A*, Dbx overwrites the instruction at *A* with an instruction that will cause a trap. The particular instruction, its length, and the kind of trap all vary across machine architectures. Before overwriting the location, Dbx copies the instruction at *A* into a local data structure. To remove a breakpoint, Dbx writes the copied value back into the target location.

Two distinct events may require setting a breakpoint at the same location. For example, suppose the user entered the following commands:

```
trace f
stop in f if x < 0
```

The first command asks Dbx to display a message whenever function *f* is called. The second command asks to suspend execution when *f* is called and the value of *x* is negative. Both requests are implemented by setting a breakpoint at the beginning of *f*. Using the implementation of breakpoints described above, the following sequence is possible:

1. Set a breakpoint for the trace command at the beginning of *f* :
   (a) Save the old instruction
   (b) Write a breakpoint instruction
2. Set a breakpoint for the stop command at the beginning of *f*:
   (a) Save the old instruction (now a breakpoint instruction)
   (b) Write a breakpoint instruction
3. Continue execution
4. Remove the breakpoint for the trace command by writing the saved instruction.
5. Remove the breakpoint for the stop command by writing the saved instruction (a breakpoint instruction!).

This sequence makes it possible to leave a breakpoint instruction at the beginning of *f* instead of the original instruction. To avoid this problem, Dbx keeps a list of all breakpoints to search when a breakpoint is set to see if the location has already been set.

Dbx sets all breakpoints every time execution continues and removes all breakpoints every time execution stops. This approach is simple but inefficient; many unnecessary accesses can occur, particularly when tracing. A better approach would be to have the instruction decoder check the breakpoint list before accessing memory.

Porting experiences

Porting Dbx to a different machine takes about one week, where roughly half the time is spent coding and the other half testing and debugging. A collection of test cases that were added for regression testing have become the driving force behind the porting procedure. A port is defined to be complete when all the tests run successfully, and the central activity during a port is to get successive test cases to work.

Every port consists of modifying Dbx to handle (1) the machine's instruction set, (2) differences in the site's version of Unix (the most common differences are in accessing process information), and (3) the calling sequence used by compilers. Dbx's organization has ''bent'' slightly with each port to handle newly-found machine dependencies, but it has never needed a major restructuring.

Regardless of how many times Dbx had been ported or how much was known about the target machine, every port included at least one surprise. Sometimes the surprises were bugs in the target system. For example, in one port the operating system crashed as a result of Dbx reading a non-existent address in the debuggee.

Usually the bugs revealed the need for new functionality in Dbx because the port was to a new machine with new compiler or operating system technology. For example, the RT-PC compiler always passes arguments to a function in registers. Inside the function, an argument value may stay in its register, be moved to a different register, or be copied onto the stack. Thus, the location of a variable's value may depend on where execution is within the program. For example, a value could be in r2 at a function entry point and on the stack after the function prolog. Because debugging traditionally has meant disabling compiler optimizations, Dbx assumed that every symbol had a statically-assigned address, stack offset, or register number. With the arrival of RISC machines, debugging (at least partially) optimized code is a requirement.

Despite problems that would have been impossible to anticipate, Dbx has been able to accommodate new features with a relatively small number of changes. In the case of register parameters on the RT, it was necessary to replace an access to the field that contained a parameter's address with a function call that computed the address based on the current program location.

Future directions

Dbx has been relatively easy to adapt to different languages and machines, but it needs additional functionality to meet the needs of more integrated programming environments and the diverse requirements of parallel, distributed, and real-time applications. At the same time, Dbx is already too big and difficult to maintain and extend. The solution is to split Dbx into several tools.

The first step is to remove functionality from Dbx that should be handled by existing programming tools. Showing source upon reaching a breakpoint or setting a new breakpoint should be handled directly by an editor. Parsing expressions should be handled by a compiler, especially when supporting languages like C++ with operator overloading and inheritance.

Integrated environments such as Smalltalka[2] combine the editor, compiler, and debugger into a monolithic system. Unfortunately, this coupling makes it difficult to support multiple editors, languages, or debuggers. Our solution to the problem of integration is to have tools communicate with other tools directly by sending messages. For example, a debugger could send an expression to a compiler, forward the generated code to a linker for relocation, load the code into an executing program, and have the program execute the code in the proper context. The overall architecture of this kind of environment is described in a separate paper [7].

For the component of debugging that handles program execution, we are building a separate application for monitoring and controlling processes. This application, called Ndb, will be a *server* that provides access to one or more processes and address spaces. Figure 5 shows an example of how Ndb might be used.

Ndb will support a standard protocol that defines operations on processes, address spaces, and breakpoints. It is critical that this protocol be simple yet powerful enough to support debugging and monitoring of sequential, parallel, or distributed applications. In particular, Ndb will *not* resolve program names or interpret language-specific expressions.

The primary advantage of a server is that several clients can share access to the same target through Ndb. For example, a performance monitor could show execution time information during a debugging session to help find performance problems. By adding new client applications, the debugging environment can be extended without changing current applications or the Ndb server. A second advantage is that the server can run on a different machine from the clients.

The implementation of Ndb can be customized to the target environment. If possible, Ndb would map the target's address space into Ndb's memory. When a request is sent to set a conditional breakpoint, Ndb would patch the test into the debuggee if sufficient memory were available; otherwise Ndb would set a trap at the point and check the condition itself.

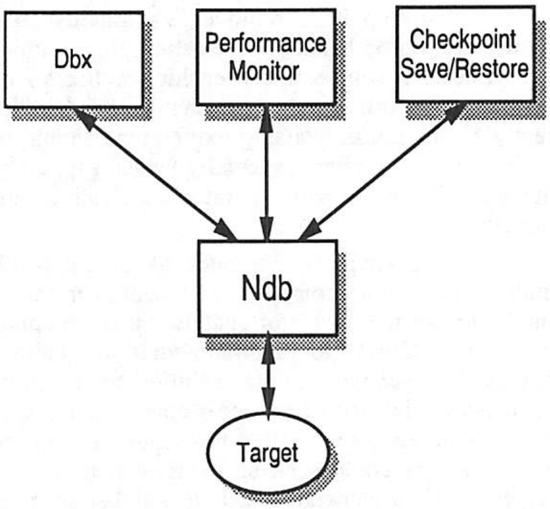


Figure 5: Using the Ndb server

Conclusions

The organization of Dbx is the key to its portability and extensibility. The relative ease in handling the surprises associated with a port demonstrates that a good design leads to portability. The converse is not true; a portable design is not necessarily good. Software designed for specific portability requirements is likely to fail because it is impossible to predict future targets to which one will want to port.

The use of stabstrings to represent type information illustrates a paradox of software systems: the best design for an product's lifetime might not be the best technical design at a particular time. For a fixed set of languages and compilers, there are many formats that are superior to stabstrings in clarity and performance. However, because of the continual changes to the environment around Dbx, stabstrings were the right choice for the long term.

In the future, Dbx should be split into several applications so that portions of its functionality can be more easily used and extended. This partitioning will expose the internal abstractions in Dbx and make it easier to incorporate debugging into an integrated programming environment. Ndb is a first step to this partitioning in the context of the execution environment.

Acknowledgments

Mike Clancy was my Master's project advisor while I worked on Pdx at Berkeley. Mike Powell contributed to the stabstrings design, especially to support Modula-2. Jim Terhorst and Mark Himmelstein helped with the IRIS and MIPS ports, respectively. Larry Breed helped me understand the compiler issues for the RT-PC port. Doug Pan is largely responsible for the design of Ndb. Thoughout Dbx's evolution, I have had many productive discussions about design and implementation with Evan Adams.

References

[1] S. Bourne and J. Maranzano, "A Tutorial Introduction to Adb", AT&T Bell Laboratories, Murray Hill, New Jersey, 1977.

[2] A. Goldberg, *Smalltalk-80: The Interactive Programming Environment*, Addison-Wesley, Reading, Massachusetts, 1984.

[3] W. Joy, S. Graham, and C. Haley, "Berkeley Pascal User's Manual", Version 2.0, EECS Department, University of California at Berkeley, October 1980.

[4] H. Katseff, "Sdb: A Symbolic Debugger", ATT Bell Laboratories, Holmdel, New Jersey, 1979.

[5] B. Kernighan and J. Mashey, "The Unix Programming Environment", *Computer*, Vol. 14, No. 4, April 1981, pp. 12-22.

[6] M. Linton, "A Debugger for the Berkeley Pascal System", Master's report, Computer Science Division, University of California at Berkeley, June 1981.

[7] M. Linton, "Distributed Management of a Software Database", *IEEE Software*, Vol. 4, No. 6, November 1987, pp. 70-76.

Mark A. Linton is an assistant professor in the Computer Systems Laboratory of the Department of Electrical Engineering at Stanford University. He received the B.S.E. from Princeton University in 1978 and the M.S. and Ph.D. in Computer Science from the University of California, Berkley, in 1981 and 1983 respectively. In addition to publishing papers on a variety of computer systems topics, he is the author of the Unix debugger "dbx", the Stanford workstation benchmarks, and most recently the C++ user interface toolkit "InterViews". His current research interests are in programming environments, user interfaces, operating systems, and workstation architectures. Reach Mark at CIS 213, Stanford, CA 94305. His electronic address is linton@interviews.stanford.edu .

USENIX Summer Conference
June 11-15, 1990
Anaheim, California

Dalek: A GNU, Improved Programmable Debugger

Ronald A. Olsson, Richard H. Crawford, W. Wilson Ho – University of California at Davis

ABSTRACT

Dalek is a debugger for UNIX systems that offers considerably more power than existing UNIX debuggers. Dalek achieves this by providing a powerful debugging language and events. The Dalek debugging language provides mechanisms comparable to those found in conventional programming languages, i.e., conditional and looping statements, blocks, local variables, procedures, and functions. The user associates arbitrary Dalek code with breakpoints in a program—e.g., to print out the contents of a binary tree or to ensure that a linked list is circular. Dalek events provide a way to form higher-level abstractions during the execution of a program. Low-level events (including primitive events such as those that can be raised at breakpoints) can be combined into higher-level events that correspond more closely to the abstractions present in a program; the programmer need not be concerned with the occurrences of the low-level events, only with their net effect as a high-level event. As an example, the low-level events of a push and a pop on a stack can be combined into one higher-level event. Dalek's method of combining events is novel: it is similar to coarse-grained dataflow, where each token corresponds to an instance of an event and nodes are used to combine low-level events into higher-level ones. Tokens can carry user-specified attributes to distinguish among different occurrences of the same event. Nodes in the dataflow graph contain Dalek code by which the user states how to combine events. Dalek's approach is more powerful than existing schemes, which typically employ pattern matching. The mechanisms Dalek provides also facilitate its use as a performance measurement tool and for patching code on the fly.

A prototype implementation of Dalek has been operational since summer 1988. It is based on *gdb*, the Free Software Foundation's GNU project debugger, and enhances *gdb*'s functionality with the additional language and event mechanisms described above. Dalek is being incorporated into a future release of *gdb*. This paper describes Dalek, its implementation, and experience using it.

1. Introduction

Existing debuggers for sequential compiled UNIX programs (e.g., *adb* [Maranzano79], *dbx* [DBX83], *sdb* [Katseff79], and *gdb* [Stallman89]) require the user to interact at a very detailed level instead of at the level of the abstractions present in the user's program.[1] Such debuggers typically provide users with a breakpoint facility, which allows them to suspend execution and examine variables when control reaches a specified point in the inferior process.

Current UNIX debuggers have two inadequacies.

- They provide only very limited control over the actions taken when a breakpoint occurs. At best, they provide only conditional execution of the entire block of commands associated with a breakpoint. More powerful and discriminating means of control are desirable, e.g., a loop to print out the contents of a linked list, or a loop to single-step the inferior process until one variable is greater than another.
- They provide the user no means, other than mentally or using pencil and paper, of correlating the occurrence of several logically related breakpoints into a single, more abstract occurrence. This capability is desirable since abstractions in an application program often require a series of procedure calls.

Our debugger, Dalek[2], remedies both of these inadequacies. Dalek's language provides conditional and looping statements, blocks, local variables, procedures, and functions. This approach is similar to

[1] The term 'user' refers to a person developing and debugging a program. The term 'debugger' refers to a program that aids the user in debugging. The debugger controls the execution of the user program, which executes as a separate, 'inferior' process.

[2] A Dalek is a semi-mechanical alien life form that was opposed by Doctor Who in its efforts to "exterminate" everything in sight [Tulloch83].

that in [Johnson78] except Dalek includes these mechanisms as part of a conventional (and modern) debugger rather than in an interpretive environment. Dalek also provides support for *events*—occurrences of interesting activities in the execution of the inferior program—as a way to form higher-level abstractions during the execution of a program. Dalek employs a novel, coarse-grained dataflow approach for combining events in which the dataflow graph's nodes contain fragments of code written in the Dalek language. Event-based debugging has received much attention for debugging concurrent programs (e.g., [Baiardi83], [Bates82], [Bates83], [Bates88], [Elshoff88], and [Lin88]), but it has not been applied to debugging sequential programs. The mechanisms Dalek provides also facilitate its use as a performance measurement tool and for patching code on the fly.

The rest of this paper is organized as follows. Section 2 gives an overview of Dalek, using several examples to illustrate Dalek's usefulness and power, and discusses design and implementation issues that were fundamental in Dalek's development. Section 3 evaluates Dalek from the perspectives of implementation effort, performance, and lessons learned. Finally, Section 4 contains some concluding remarks, including the direction our work on debugging is taking.

2. The Dalek Solution

In this section, we introduce Dalek and demonstrate by means of examples how it can be used to solve common debugging problems. We also discuss related design and implementation issues. (See [Crawford89,Crawford90] for additional details, further examples, and other goodies.)

Overview of Dalek

Dalek is based on *gdb* (version 3.0). It enhances *gdb*'s functionality in two significant ways. Dalek extends the original *gdb* language with a conditional statement (not to be confused with *gdb*'s conditional breakpoint command), a looping construct, blocks, local convenience variables[3], procedures, and functions. It also supports events, an abstraction mechanism used to construct models of the inferior program's behavior (either its intended or undesirable forms) and to monitor the program's actual behavior in terms of the specified forms.

Dalek inherits many of its commands and much of its syntax from *gdb*. The following are important for this paper. The **silent** command suppresses output announcing that a breakpoint has occurred. The **cont** (short for 'continue') command resumes execution of the inferior process. The **step** command executes the next statement of the inferior process. Names of convenience variables and Dalek's built-in and user-defined functions start with a '$'. Names of events and their attributes start with a backquote, e.g., 'foo. Comments begin with a '#'.

Dalek Programming

Dalek code can be specified dynamically as part of the debugging activity, without requiring the source code to be modified, recompiled, and relinked. Besides avoiding the overhead of those activities, this approach has the additional advantage that the user can specify what is of interest as s/he gleans information during the inferior's execution, i.e., for "program exploration".

Dalek code can be executed from the debugger's command level or automatically at a breakpoint or when an event is triggered. It can be entered interactively or read from a file.

The Dalek language contains all the mechanisms that programmers generally find useful: assignment statements, **if** and **while** statements, blocks, local convenience variables, procedures, and functions. The Dalek code and variables reside within the debugger's address space. Thus, storage of these objects does not affect and is not affected by the inferior process. To illustrate the Dalek language, consider the Dalek code shown in Figure 1.

```
# Dalek procedure to print out value
# field of nodes in a circularly
# linked list starting at $head.
# Assumes $head is a dummy node, whose
# next field points to itself if the
# list is empty.

function $print_list($head)
    push-block
    create-local $p
    set $p = $head->next
    while ($p != $head)
        printf "%d\n", $p->value
        set $p = $p->next
    endwhile
    pop-block
end
```

Figure 1: A Dalek procedure to print a linked list

The code defines a $print_list() procedure. The commands **push-block**, **pop-block**, and **create-local** are used together to effect a local scope. This procedure can then be invoked interactively from Dalek's command level or from any point during execution of the inferior process by simply setting a breakpoint whose commands include a call to the procedure. As a simple example, $print_list() can be invoked from a breakpoint at line 27 of file *goo.c* by the commands

[3] A convenience variable is a variable that the user defines and manipulates within the debugger; *gdb* provides global convenience variables.

```
break goo.c : 27
commands
# suppress default announcements
    silent
# print list headed by symtab_head
    call $print_list(symtab_head)
# continue inferior process execution
    cont
end
```

Note that the body of $print_list() could have been written inline as commands in the above. However, it is written as a function so it can be invoked directly from the command level or from within another breakpoint, if desired.

In a manner similar to the above $print_list() example, Dalek code can be written to ensure that the inferior maintains certain invariants regarding its data structures. For example, code to check that a circularly linked list is indeed circular could be invoked on entry to and exit from each procedure that manipulates the list. This approach is more flexible than using compiled-in assertion checks (e.g., like the *assert* macro in C) since what invariant to check and where to check it can be decided when actually debugging the inferior.

To demonstrate further how Dalek's mechanisms can be used to attack a problem, we describe our experience using Dalek to find a bug in code generated by lex(1). The execution of the lex-generated code resulted in a segmentation fault occurring in a reference to a table element within the procedure *yylex*. Setting a breakpoint at procedure *yylex* and reexecuting the inferior showed that the procedure was a frequently executed one. We then used a counter to determine the particular invocation of *yylex* that resulted in the segmentation fault. Next we modified the breakpoint code so it would simply continue execution of the inferior for all invocations of *yylex* except the last one. For the last invocation, it would single step the inferior as long as the reference that was causing the segmentation fault was within the bounds of the table. At this point, the Dalek code looked something like that shown in Figure 2. Executing the inferior in conjunction with this debugging code thus showed the first point at which the condition went out of range. From examining the inferior's code, it was then easy to see that it was using a byte (char) for indexing a large table, and the byte had overflowed.

The iterative technique used above to isolate a bug is not new. What is interesting, however, is that Dalek gives the right tools for assisting the user in this process. This problem would be more difficult to attack using other debuggers since they do not provide **while** statements and/or convenience variables. Using *dbx*, for example, variables can be traced so that information is output when their values change; further, that output can be conditional. However, the amount of output that could be printed before the relevant output appeared would most likely overwhelm the user.

Dalek Events

An event in Dalek is the occurrence of an interesting activity in the execution of the inferior program. Each event can have any number of dynamically-typed attributes associated with it. The definition of a *primitive* event specifies a list of its attributes' names and a command block written in the Dalek language. The definition of a *high-level* event is similar to that of a primitive event but it also specifies a list of *constituents*—lower-level events whose attributes it can access and upon which its activation may depend. As with other Dalek language commands, event definitions can be given

```
set $count = 0
break lex.yy.c : yylex
commands
    silent # suppress the default announcement of encountering a breakpoint
    if( ++$count < $N )  # $N was determined earlier
        cont # continue execution of the inferior process
    else # single step the inferior while valid reference.
        #       note:  yyt->advance is used in the source statement
        #       yystate = yyt->advance + yysvec
        while( yyt->advance >= 0 )
            step
        endwhile
        printf "%d before start of table\n", yyt->advance
    endif
end
```

Figure 2: Dalek code for isolating the cause of a segmentation fault

interactively or read from a file.

Dataflow-based Event Recognition

In most models of dataflow, tokens flow between the nodes of a directed graph. In Dalek, the leaves of this graph represent primitive events (independent sources of tokens) and the interior nodes represent higher-level events. Initially, activity within the inferior process triggers a primitive event, which, if it successfully completes its recognition code, generates a token. Since a token corresponds to an *instance* of an event, a token is a composite object; the information required to characterize a particular occurrence of an event is carried with it as its attributes. Recognition of a primitive event causes copies of the token characterizing the event to flow along dedicated arcs to all high-level events that depend on it.

In the dataflow model, computation is performed at the nodes. In Dalek, the command blocks associated with events correspond to nodes. Incoming arcs are represented as queues to which arriving tokens append themselves. The definition of a high-level event specifies which of its constituents will trigger that high-level event. That is, the definition indicates which tokens, when arriving on incoming arcs, will activate the event's command block so it can attempt to recognize an instance of that high-level event. (A minus sign prefixing a constituent in the definition of the high-level event indicates that the arrival of that constituent should *not* trigger the high-level event.)

When a high-level event is triggered, its associated code decides how it should react based on what constituents are present and their values. The code might decide that the present constituents (or some subset thereof) comprise a valid instance of the high-level event. In such a case, the code should assign values to its own attributes, either by copying those of selected constituents or by computing some function based on the available data. Other possible actions include printing a message, reestablishing interactive control while keeping the inferior process suspended, and even modifying data or control flow in the inferior process. The code will then generally remove the selected constituents from its incoming queues so as to avoid repeated recognition of the same tokens. When the high-level event's code finishes execution, tokens embodying the new event instance propagate to all high-level events that depend on it, and the recognition phase begins anew. On the other hand, the code might decide that the appropriate combination of constituents is not present on the incoming queues or that their attributes fail to satisfy the desired relationships. To suppress (the normal) generation of tokens representing this event and the subsequent passing of those tokens to the higher-level events that depend on it, the code can execute an **event dont-propagate** command.

Generally, the code associated with an event executes in its entirety before another event's command block is executed. If a single event triggers multiple higher-level events, the order in which they execute is undefined. When all triggered nodes have executed—i.e., the graph reaches a quiescent state—Dalek continues execution with the command following the one that raised the initial primitive event.

The exception to the above general rule arises when a node executes a command that resumes the inferior process. For example, execution of a single step command suspends the event's code and returns control to the inferior process. The suspended event's code is then resumed after the inferior process executes one statement. Note that resuming execution of the inferior process might cause it to encounter another breakpoint, which can cause another primitive event to be raised. Dalek handles such events by stacking their executions; the details are described in [Crawford90].

A history list records all recognized event occurrences. It may be browsed selectively by the user in interactive mode or accessed programmatically via the Dalek language.

Primitive Events

In order for an instance of a primitive event to occur, it must be explicitly raised. Typically, the user specifies an **event raise** command in the command block of a breakpoint inserted in the inferior process.

If desired, a given primitive event may be raised from several different breakpoints in the source code. At the other extreme, a primitive event need not be bound to any breakpoint in the inferior program.

Primitive events can be thought of as characterizing interesting changes in the control and data state of the inferior process. Typical primitive events include entry to or exit from procedures or smaller blocks of code, as well as changes in a program's data ranging from a change in a variable's value to more complex changes such as a linked list becoming empty, an array becoming unordered, or a graph becoming cyclic. Such complex changes, though, are most often expressed as high-level events.

Since each instance of an event has its own attributes (located in Dalek's address space), and since each invocation of an event's command block can have local variables as well, the commands for each event can be written in isolation from those for other events. This structure allows events to mirror the block structure of an inferior program quite naturally. It also permits stepwise refinement of a behavioral model. Thus, the user can define events covering only those abstractions in the program that either highlight a bug's symptoms or that are suspected of causally contributing to bugs.

Using Events to Monitor Memory Allocation

To illustrate events, we examine the problem of monitoring allocations and deallocations of memory, i.e., *malloc* and *free* in C terminology. A common programming error is to pass an invalid address for deallocation, especially one for a block that was previously freed. Here we show how errors in the use of the memory allocator can be detected.

Figure 3 shows the definition of two primitive events, 'malloc and 'free, which correspond to invocations of the allocation and deallocation routines. 'malloc records (using **event set-attribute**) in its attribute 'addr the address of the memory block allocated; 'free records in its attribute 'addr the address of the memory block to be deallocated. $quiet_finish() is a Dalek built-in function that resumes execution of the inferior process until it returns from the current procedure. $quiet_finish() returns the procedure's return value. $frees_arg is a global convenience variable set at the breakpoint inside *free* to the address of the block being freed (i.e., *free*'s argument). Since *free* is a system-dependent library routine, how $frees_arg is set is also system-dependent.

Error detection is carried out in the high-level event 'match. 'match keeps track of the addresses of all memory blocks that have been allocated but not yet matched with a deallocate request. Whenever 'free occurs, 'match checks to see if the corresponding 'malloc event has been raised. If such an address cannot be found, 'match prints an error message. Note that the order of freeing blocks need not be the same as that in which they were allocated.

As shown, 'match is triggered only by 'free. The matching of 'malloc and 'free events is performed using a simple while loop, which sequentially compares the attribute 'addr passed by 'free with those returned by the previous 'malloc events.

```
# The 'malloc event is recognized each time it is triggered/raised
# by default; the same applies to the 'free event.
event define 'malloc ('addr)
    event set-attribute 'addr $quiet_finish()
end

event define 'free ('addr)
    event set-attribute 'addr $frees_arg
end

# The high-level event 'match depends on lower level events 'malloc and 'free
# but it is triggered only by 'free.
event define 'match ()   -'malloc  'free
   push-block
   create-local $qindex
   create-local $free_addr
   set $qindex = $qlen('malloc)  # $qlen gives length of specified event queue
   set $free_addr = $value('free, 1, 'addr)
   while ($qindex > 0)
      if ($value('malloc, $qindex, 'addr) == $free_addr)
         event remove 'free, 1 # remove first token from queue for 'free
         event remove 'malloc, $qindex  # and $qindex'th token for 'malloc
         wbreak  # break out of the while loop
      endif
      set $qindex--
   endwhile
   if ($qindex == 0)
      printf "Error: attempt to free unallocated address 0x%x\n", $free_addr
      event remove 'free, 1
      event dont_propagate
   endif
   pop-block
end
```

Figure 3: Memory allocator monitor using Dalek events.

This example can be extended to determine whether the reason for a mismatched *free* is that the block is being freed a second time. To do so, 'match requires only small modifications. An attribute, say 'maddr, needs to be specified in its definition and set when a match is found. Also, a loop needs to be added to the body of the second **if** statement. The loop simply searches the event history list for a 'match whose 'maddr attribute is equal to $free_addr. With additional attributes in 'malloc and 'free recording the names of their calling functions, we can make the crucial transition from merely detecting the symptoms of a bug to discovering the location of its cause.

Searching the history list is also useful in different contexts. Suppose, for example, we detect that the contents of an allocated structure have been corrupted. At that point, the user can type in a while loop to search the history list to determine whether the address of that structure was ever freed.

Using Dalek to Measure Performance

Dalek can also serve as a tool for measuring run-time performance of the inferior process. In particular, it can be used to time parts of a program and to profile sections of code. A considerable advantage of using Dalek in these roles is that the part of the program on which information is to be gathered, as well as the precise information to be gathered, can be specified interactively and dynamically, without modifying, recompiling, and relinking the inferior program's source.

Dalek provides three built-in functions that return the cpu time used by the inferior process. $utime() returns the inferior's user time, $stime() returns the inferior's system time, and $ttime() returns the inferior's total time. (User time is the time the cpu spends executing the instructions in the user process's address space, whereas system time is the time the cpu spends executing system calls invoked by the process; total time is the sum of user time and system time.) Unfortunately, only $utime() is accurate. $stime(), and therefore $ttime(), is not accurate since some of the cost of the debugger controlling the inferior process is charged as system time to the inferior process. These functions can be used in the code that is executed as part of a breakpoint or event, and therefore can easily be changed interactively and dynamically.

One use of Dalek's timing functions is for tuning a program. For example, suppose we want to determine the times for both successful searches and unsuccessful searches of a symbol table. This information can give us feedback on the appropriateness of our data structure representation, perhaps suggesting that a hash table would perform better than a linked list. To obtain these times, one breakpoint can be set on entry to the search function and another breakpoint can be set right before the search function returns. The first breakpoint records the starting user time; the second adds the difference between the current user time and the recorded starting user time to the running sum for either successful searches or unsuccessful searches according to the value the search function is returning. Alternatively, the breakpoints could raise events that include the time information as attributes. That would allow a more detailed analysis, say to compute averages and variances, since each instance of an event is recorded on the event history list.

Gathering timing information using Dalek, instead of using tools like gprof(1) or including timing code in the source program, provides the user with more control over what is being measured and is not intrusive. As illustrated in part by the above example, individual or groups of procedures or statements can be measured, using arbitrary conditions to select those of interest. gprof does not provide such fine-grained control. Including timing code in the source program is viable, but that requires recompilation and can affect overall user time measurements. The number and kinds of changes to the source code required by that approach can also become unwieldy. For example, suppose we want to find the cumulative time spent executing in procedure P for those invocations from procedure Q. To do such measurements by modifying the source code requires either an additional parameter in procedure P's interface or a global variable; either of these changes is cumbersome in a program of any size. In Dalek, the name of the calling procedure N levels up in the stack frame can be determined using $func(N).

Dalek can also be used as a simple code profiler and to gather other useful statistics concerning program behavior. For example, Dalek has been used to count the number of times a given procedure or statement is executed, to find the maximum length a linked list reaches during execution, and to measure run-time stack frame behavior in order to determine conditional probabilities that a push follows another push, a pop follows a push, etc. This kind of information is easily obtained by setting breakpoints at appropriate points in the code and specifying breakpoint commands to maintain convenience variables; **if** statements and functions are used for performing the more complicated measurements.

On the Fly Code Patching

Dalek can also be used for "code patching", in which original (e.g., buggy) program code can be effectively replaced, during debugging, with code written in Dalek. This kind of patching is temporary, affecting only how the inferior executes while under control of the debugger; it does not modify the program's executable file or the source code. Using such code patching, multiple errors can be found and fixed without exiting the debugger and recompiling the program. For example, suppose that debugging indicates that

```
symbol[end-start] = '\0';
```

which appears as line 38 in the file symtab.c should really be

```
symbol[end-start+1] = '\0';
```

Then, a breakpoint can be placed at that line whose (Dalek) code simply executes the correct statement and then jumps around the bad code. That is,

```
break symtab.c : 38
commands
# suppress default announcements
    silent
    set symbol[end-start+1] = '\0'
# skip over the bad statement;
# continue execution of the inferior
    jump 39
end
```

Each time the inferior reaches that line, the commands associated with this breakpoint are executed instead of the original, buggy code. Using techniques similar to the above, new (Dalek) code can be added to a program to initialize variables, etc. It is also easy to replace entire functions or groups of statements since the Dalek language provides conditional and looping statements and functions.

Design Decisions and Tradeoffs

Our main goal in designing Dalek was to focus on the fundamental problems in debugging and to offer solutions to those problems. One early decision we made was not to expend effort developing a graphical interface. While such an interface would be useful, we chose instead to work on what we consider to be the more basic problems.

Dalek is intended to work on existing executables, preferably compiled with the '-g' flag so that Dalek can obtain the executable's symbolic names, types, and line numbers. Furthermore, Dalek requires no modifications to the compiler or program's source code. In fact, Dalek encourages a very interactive, dynamic style of debugging, e.g., as illustrated above with the code patching and measurements examples. This approach allows the user to explore a program during its execution, adding routines as desired. Injecting such debugging code into the source code, perhaps by conditional compilation, is not an attractive solution. Often, the user has arduously brought the inferior process to a critical point in its execution under the debugger's control before discovering the need for such routines. Furthermore, seemingly minor modifications to the source code can substantially alter the symptoms exhibited by bugs, especially if pointers are involved.

The Dalek language provides all the mechanisms that programmers generally find useful: assignment statements, **if** and **while** statements, blocks, local convenience variables, procedures, and functions. Dalek code can be written when the user finds it necessary, e.g., when execution of the inferior process is suspended at a breakpoint. Furthermore, the same language is used at the debugger's command level and within code written for breakpoints and events. The Dalek code and variables reside within the debugger's address space. Thus, storage of these objects does not affect and is not affected by the inferior process.

Providing **if** and **while** statements within the debugger language means we had to resolve language semantic and implementation issues for combinations of features. For example, single-stepping the inferior from within a while loop normally means that the next debugging command to be executed is the next one in the while loop, unless that step causes the inferior to execute another breakpoint. Thus, Dalek's implementation needs to maintain a stack of pending command blocks.

The notion of event-based debugging is common in the debugging of concurrent programs (e.g., [Baiardi83], [Bates82], [Bates83], [Bates88], [Elshoff88], and [Lin88]). However, it has not been applied to debugging of sequential programs. Our approach to event recognition differs considerably from other such approaches. Other event-based approaches typically use a special notation (e.g., based on pattern matching) to specify how lower-level events should be combined into higher-level ones. Our approach, on the other hand, uses dataflow and programmable nodes for that purpose. Our approach is intentionally low-level to give the user flexibility and to allow us to explore the capabilities of our prototype. Unlike other approaches, it does not constrain event recognition by preconceived notions of how higher-level events will be formed. One drawback of our approach, however, is that even common patterns need to be explicitly programmed. We will remedy that, after gaining further experience with Dalek, by providing a macro facility or libraries.

One final interesting design decision we made was to provide "broadcast breakpoints": breakpoints set en masse at the entry to every procedure whose name matches a given regular expression. Such breakpoints can share a common block of breakpoint commands. Simple procedure tracing, for example, can be effected by issuing a single broadcast breakpoint command with "*" as the regular expression and

```
silent
# output executing procedure name
print $func(0)
cont # continue execution
```

as the command block. The command block uses the built-in Dalek function $func() to obtain the name of the function associated with the current frame. Another form of broadcast breakpoints associates a breakpoint with each instruction in a given address range whose opcode matches the specified pattern.

Implementation Issues

Dalek's implementation is based on *gdb*'s. That, of course, saved us considerable time. However, the support for some of Dalek's high-level features required some innovation. Two particularly interesting implementation issues involve broadcast breakpoints and garbage collection.

Broadcast Breakpoints

Recall that a command establishing a broadcast breakpoint can specify many procedures at which to place a breakpoint. Execution of a program with a large number of breakpoints under *gdb*, however, can be very slow. Specifically, *gdb* removes *all* breakpoints from the inferior's code space when control passes from the inferior to the debugger; it re-inserts them when control passes back. Removing and re-inserting each breakpoint requires two calls to ptrace, each of which requires the UNIX kernel to perform two context switches.

Dalek avoids this cost unless the inferior actually executes a breakpoint. The one potential problem with this solution is that Dalek could display incorrect assembly code if it displays exactly the instructions it reads from the inferior's code space since those might include the breakpoint instructions. That would typically lead to "framing errors", causing subsequent instructions to be incorrectly displayed. Dalek avoids this problem by intelligent use of *gdb*'s existing breakpoint data structures to show the original code (i.e., without the breakpoint instructions).

Garbage Collection

gdb uses a very simple garbage collection algorithm. As it executes each command, it links together on a "freeable" list all memory that it allocates. If the contents of a node might be needed later (e.g., when a new convenience variable is assigned) it removes the node from the freeable list. After executing each command, *gdb* frees all nodes on the freeable list.

Dalek's high-level debugging features, however, require a more complicated garbage collection scheme. As one example, Dalek's functions can return values that can be used in expressions. Their values cannot be freed, as *gdb* normally would, until after they are used in the evaluation of the rest of their enclosing expression. Our implementation, therefore, delays freeing an object until the command that created it has been exited with no possibility of resumption. In essence, we maintain the freeable list as a stack.

3. Evaluation

Implementation Effort

We have implemented all of the Dalek features described in this paper and several other features. (See [Crawford90] for more details on Dalek's implementation.) Our implementation augments *gdb*'s (version 3.0) approximate 35,800 lines of C source code by roughly 9,400 lines. Development activities took about one person-year.

The major effort in implementing Dalek was spent

- developing support for **if** and **while** statements, procedures, and blocks with local variables;
- developing support for events;
- and getting these features to work well together and with the rest of *gdb*'s features.

Many of the required changes were straightforward. The changes for the **if** and **while** statements, for example, required adding new routines for interpreting those statements. Other implementation changes, such as dealing with stacks of pending commands and needing a better garbage collection algorithm mentioned earlier, required more implementation effort. The part of Dalek's implementation that deals with events and the dataflow engine is fairly independent of the rest of the implementation. This part did, however, require several "hooks" into the scanner so that names of attributes, for example, were recognized as such.

One desirable feature that has not been implemented is to allow new types to be defined within Dalek. That is, convenience variables are restricted to have types that have been declared in the inferior process.

Performance Data

The memory allocation example presented earlier was actually used to find an insidious memory allocation bug in Dalek itself. That is, we used Dalek to debug itself. Using Dalek to monitor all the memory allocation and deallocation for a large complicated program like Dalek took a fair amount of time. Dalek required about one minute of elapsed time on a moderately-loaded (load average about 6) VAX 8600 before the inferior Dalek finished initializing its data structures; the overhead for the rest of the monitoring, while noticeable, was acceptable. In that one minute, Dalek handled 399 events generated by 369 *mallocs* and 15 *frees*; i.e., 369 'malloc, 15 'free, and 15 'match events were triggered. Using Dalek for such debugging is certainly more convenient and much faster than manually checking the arguments in each invocation of *malloc* and *free.* The two major cost factors in this example are in context switching between Dalek and the inferior process (two for each ptrace system call) and in interpreting Dalek commands.

Dalek's implementation is currently far from optimal. For example, each command inside a while loop is parsed from its source text on each iteration of the loop. This behavior fits better with *gdb*'s existing structure. It also affects the performance of Dalek's event handling since commands in event handling code are interpreted in the same manner.

What would we do differently?

Dalek was developed as a prototype to allow us to explore our ideas in debugging. As such, we decided that it would be easier to extend an existing debugger than it would be to develop an entirely new debugger from scratch. We chose to use *gdb* as our starting point because its source is readily available, it runs on a wide range of architectures, and its command language is fairly clean both syntactically and semantically. While *gdb* gave us a very good platform on which to build, it also caused us some problems, for example, due to the integration of non-compatible features.

One such problem is that *gdb* parses its input in a line-oriented manner. That constrained our syntactic options in designing the Dalek language and explains why we have commands like **push-block**, **pop-block**, and **create-local** to effect a local scope instead of more standard (and palatable) syntax. Also, as mentioned above, commands are parsed each time they are encountered, which has an adverse affect on Dalek's performance. Translating the commands into an internal representation would considerably improve Dalek's performance.

A second problem we had building Dalek on top of *gdb* is related to breakpoints. When single-stepping under *gdb* on some architectures (e.g., Sun 3's), if a breakpoint instruction is encountered, the breakpoint is not detected and the associated commands are not executed. That behavior may be reasonable in *gdb* but is problematic in Dalek since the breakpoint might be associated with raising an event or used to transfer control to a code patch.

Ideally, we would have started the language and implementation from scratch. In fact, we are considering a new version of Dalek whose implementation would continue to use *gdb*'s low-level routines for managing the inferior, but would use an entirely new command interpreter that would support free form input and parse commands into an internal representation, as suggested above.

One item on our wishlist is additional operating system and architectural support for debugging. For example, the only standard UNIX system call to support debugging is ptrace. As mentioned earlier, ptrace is very primitive and expensive, requiring two context switches for each word of data transferred between the debugger and the inferior on some machines (e.g., a VAX). The kind of high-level debugging encouraged by Dalek can require a large amount of information to be passed between the debugger and the inferior. For example, event attributes are often values of variables in the inferior, which need to be copied into Dalek's space when an event is raised. Additional support for debugging could speed up the high-level debugging we describe here. For example, allowing one process (the debugger) to read directly the address space of another (the inferior) can greatly reduce the number of system calls. That kind of facility has been implemented by treating processes as files [Killian84]; e.g., System V Release 4 UNIX systems provide the "/proc" file system. Another approach is to allow the debugger to share some of the inferior process's address space [Aral88].

4. Conclusion

This work is significant for two reasons. First, it defines mechanisms for debugging at a higher level than existing debuggers. In particular, Dalek's debugging language provides: a full repertoire of conventional programming features (e.g., local variables, conditional and looping statements, and procedures); ways to define, raise, and recognize events; and ways to subsequently search and correlate events in the "event database", i.e., the event history list. The examples demonstrated the utility of these mechanisms in solving real debugging problems. Second, this work demonstrates that those mechanisms are feasible by incorporating them into a working prototype, i.e., Dalek. Having the prototype has given us an opportunity to obtain valuable feedback on our ideas and their implementation, and a powerful debugging tool that we and others find most useful. Having the prototype has also encouraged us to experiment with Dalek's use as a dynamic performance measurement tool, as described earlier. Dalek is being incorporated into a future release of *gdb*.

Dalek's execution time for high-level debugging is in general reasonable, and certainly much less than would be required for a user to do the same work manually. For example, for a given problem that would require several person-hours employing other debuggers, Dalek might require minutes of computer time. In some cases, however, Dalek's execution time can be high. As mentioned earlier, additional operating system and architectural support for debugging would drastically reduce costs in those cases.

Our future work will include working on Dalek language mechanisms, studying further ways to optimize Dalek's implementation, and generalizing our approach to allow debugging of concurrent programs.

Acknowledgements

Many thanks are due to Richard Stallman and the Free Software Foundation for making *gdb* available to the public. Having such a good base on which to build Dalek greatly expedited our work. Jim Kingdon of FSF is integrating Dalek back into *gdb*. Mudita Jain, Carole McNamee, Chris Wee, and Cui Zhang provided very useful comments on Dalek and on earlier drafts of this paper. Early work on this project was supported in part by System Integrators, Incorporated (SII) and the State of California under the MICRO program.

References

[Aral88] Z. Aral and I. Gertner. ''Non-Intrusive and Interactive Profiling in Parasight''. *Proc. of the ACM/SIGPLAN Parallel Programming: Experience with Applications, Languages and Systems*, 21-30, July 1988.

[Baiardi83] F. Baiardi, N. De Francesco, E. Matteoli, S. Stefanini, and G. Vaglini. ''Development of a debugger for a concurrent language''. *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, Pacific Grove, CA, 98-106, March 1983.

[Bates82] P. Bates and J. C. Wileden. ''EDL: A basis for distributed system debugging tools''. *Proc. Fifteenth Hawaii International Conference on System Sciences*, Honolulu, HI, 86-93, 1982.

[Bates83] P. Bates and J. C. Wileden. ''An approach to high-level debugging of distributed systems (preliminary draft)''. *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging*, Pacific Grove, CA, 107-111, March 1983.

[Bates88] P. Bates. ''Debugging heterogeneous distributed systems using event-based models of behavior''. *Proc. Workshop on Parallel and Distributed Debugging*, Madison, WI, 11-22, May 1988.

[Crawford89] R. H. Crawford, W. W. Ho, and R. A. Olsson. A Dataflow Approach to Event-Based Debugging. CSE-89-7, Div. of Computer Science, University of California at Davis, May 1989.

[Crawford90] R. H. Crawford. Topics in Behavioral Modelling and Event-Based Debugging. M. S. Thesis, Div. of Computer Science, University of California at Davis, in preparation.

[DBX83] ''DBX (1)'' in *UNIX Programmer's Manual*, 4.2 Berkeley System Distribution, Vol. 1, Computer Science Division, University of California, Berkeley, CA, August 1983.

[Elshoff88] I. J. P. Elshoff. ''A distributed debugger for Amoeba''. *Proc. Workshop on Parallel and Distributed Debugging*, Madison, WI, 1-10, May 1988.

[Johnson78] M. S. Johnson. ''The Design and Implementation of a Run-Time Analysis and Interactive Debugging Environment'', Ph.D. Dissertation, The University of British Columbia, TR-78-6, August 1978.

[Katseff79] H. P. Katseff. ''Sdb: a symbolic debugger''. in *UNIX Programmer's Manual, 7th Edition, Vol. 2C* Bell Laboratories, Holmdel, NJ, January 1979.

[Killian84] T. J. Killian. ''Processes as files'', Proc. of the Summer 1984 Usenix Conference, 203-207, June 1984.

[Lin88] C. Lin and R. J. LeBlanc. ''Event-based debugging of object/action programs''. *Proc. Workshop on Parallel and Distributed Debugging*, Madison, WI, 23-34, May 1988.

[Maranzano79] J. F. Maranzano and S. R. Bourne. ''A tutorial introduction to ADB''. in *UNIX Programmer's Manual, 7th Edition, Vol. 2A* Bell Laboratories, Holmdel, NJ, January 1979.

[Stallman89] R. M. Stallman. *GDB Manual (The GNU Source-Level Debugger), Third Edition, GDB version 3.1*, Free Software Foundation, Cambridge, MA, January 1989.

[Tulloch83] J. Tulloch and M. Alvarado. *Doctor Who—The Unfolding Text*, Macmillan Press, 1983.

Ron Olsson received the B. A. degree in mathematics and in computer science, and the M. A. degree in mathematics, from the State University of New York, College at Potsdam. He received the M. S. degree in computer science from Cornell University and the Ph.D. degree in computer science from The University of Arizona. Dr. Olsson has been assistant professor of computer science at the University of California at Davis since 1986. His research activities center around concurrent programming languages—focusing on issues in design, implementation, optimization, and debugging—and system software. Reach him electronically at olsson@ivy.ucdavis.edu.

Rick Crawford earned his B. S. in Mechanical Engineering from the University of California at Berkeley in 1979. After receiving firsthand experience in the "real world" regarding how large software projects should not be managed, he sought sanctuary amidst the relative sanity of academia. He is currently a graduate student in computer science at the University of California at Davis. His programs (except for Dalek) are always bug free. Reach him electronically at crawford@ivy.ucdavis.edu.

Wilson Ho is a graduate student and Ph.D. candidate in the Department of Electrical Engineering and Computer Science at the University of California at Davis. He received his B. S. in computer science from the University of Hong Kong in 1986, and his M. S. in computer science from the University of California at Davis. His research interests include debugging of distributed programs, distributed file systems, and programming environments. His programs (except for Dalek) are never bug free. Reach him electronically at how@ivy.ucdavis.edu.

USENIX Summer Conference
June 11-15, 1990
Anaheim, California

The Design of a Secure Internet Gateway

Bill Cheswick – AT&T Bell Laboratories

ABSTRACT

The Internet supports a vast and growing community of computers users around the world. Unfortunately, this network can provide anonymous access to this community by the unscrupulous, careless, or dangerous. On any given Internet there is a certain percentage of poorly-maintained systems. AT&T has a large internal Internet that we wish to protect from outside attacks, while providing useful services between the two.

This paper describes our Internet gateway. It is an application-level gateway that passes mail and many of the common Internet services between our internal machines and the Internet. This is accomplished without IP connectivity using a pair of machines: a trusted internal machine and an untrusted external gateway. These are connected by a private link. The internal machine provides a few carefully-guarded services to the external gateway. This configuration helps protect the internal internet even if the external machine is fully compromised.

Introduction

The design of a Corporate gateway to the Internet must deal with the classical tradeoff between security and convenience. Most institutions opt for convenience and use a simple router between their internal internets and the rest of the world. This is dangerous. Strangers on the Internet can reach and test every internal machine. With workstations sitting on many desks, system administration is often decentralized and neglected. Passwords are weak or missing. A professor or researcher often may install the operating system and forget it, leaving well-known security holes uncorrected. For example, a sweep of 1,300 machines inside Bell Labs around the time of the Internet Worm found over 300 that had at least one of several known security holes.

When we first obtained a connection to the ARPAnet, Dave Presotto configured our gateway machine (named arpa) as an application-level gateway. For two years this machine was the sole official link to the Internet for AT&T. Until its disconnection a little while ago, this VAX 750 handled all the Internet mail traffic and other services for the company. Arpa had Ethernet connections to both the inside and outside Internets, just like a router. It could also make and accept calls on our corporate Datakit network.

Dave took a number of steps to make our gateway more secure. He turned off IP forwarding in the kernel so packets could not travel between the Internets. He installed a kernel modification that limited TCP connections from arpa to the inside network to *smtp, uucp, named,* and *hostname* ports. And he rejected the *sendmail* mailer as too complicated and dangerous: the Upas[upas] mailer was installed in its place. We removed a number of non-essential daemons, including the *finger* server.

To give insiders access to the Internet, a *gate* service was installed on arpa. Insiders could call this service and supply an Internet address. The gate connected to a socket of a remote Internet host and then copied bytes between the two connections. It was easy to provide *atelnet*, a version of *telnet* that used the gate service. *Aftp* supplied FTP services: it was the standard FTP modified so both the command and data connections were initiated from the inside. (The standard *ftp* would have tried to make the data connection from arpa to the inside, a connection prohibited by arpa's kernel.)

This configuration successfully resisted the Internet worm. We ran neither *sendmail* nor *fingerd*, the two programs exploited by the worm.[seeley] The internal internet was spared the infection. (Actually, there was a second, unguarded IP link to the Outside. We got lucky: only a few machines at the other end knew of the link, and their machines were shut down before the worm could creep across.)

Had arpa been infected, the worm could have reached the inside machines. The initial *smtp sendmail* connection was permitted, and the worm's second connection would have been initiated from the inside target machine into arpa, the permitted direction.

The new gateway

All of arpa's protection has, by design, left the internal AT&T machines untested—a sort of crunchy shell around a soft, chewy center. We run security scans on internal machines and bother system administrators when holes are found. Still, it would be nice to have a gateway that is demonstrably secure to protect the internal machines. For peace of mind, the gateway design should not rely on vendors' code more than absolutely necessary. We would like the internal machines protected even if an invader breaks into the gateway machine, becomes root, and creates and runs a new kernel.

We had to replace arpa. The VAX 750 ran with typical load averages of seven to twelve jobs throughout the day. When the load average hit about fifteen, the old Datakit driver expired, wedging the Datakit ports and requiring a reboot.

A new machine gave the opportunity for a clean start. We could re-think the security arrangements to improve on arpa's shortcomings.

Our new gateway machine, named inet, is a MIPS M/120 running System V with Berkeley enhancements. Various daemons and critical programs have been obtained from other sources, checked, and installed.

We store nothing vital or secret on inet, since we assume that it may be defeated in unforeseen ways. It does not currently run *uucp*—systems files and dialers could fall into the wrong hands. There are few system administration accounts, and user accounts are discouraged. Inet is not used for other tasks. It is backed up regularly, and scanned for unauthorized changes and common system administration mistakes. Though we don't trust inet, we protect it as much as we can.

Inet has a single Ethernet port which is connected to a router on JVNCnet, our external regional network. It also has a connection to Datakit. We have configured our Datakit controller to force all connections from inet to a single internal machine, named r70. R70 can redial, or splice connections to other internal machines. R70 provides a limited set of services to inet for reaching internal machines. The list of services is:

1. connection to an approved machine's *smtp* port,
2. connection to a login or trusted-login Datakit destination after passing a challenge-response test, and
3. connection to a logging service.

The key to the arrangement is a restricted channel from inet to r70. This private channel was easily constructed using stock features of our research Datakit controller. Other connection schemes could be implemented using a simple multiplexed protocol over some back-to-back connection between the machines, or a simple Ethernet would suffice. If the last approach is used with TCP, the internal machine must supply differing TCP services to its two Ethernet interfaces. (I am not sure this is possible with standard TCP/IP implementations. It wouldn't be too hard to modify *inetd* to do this.)

These functions do not load the internal machine too much; it could have other uses like *uucp*, *mail*, or even normal user jobs. But the services it provides the external machine are the key to security, and must be protected well.

Outbound service

It is quite easy to implement most outbound services to the Internet. Inet has a small program, named *proxy* (a descendant of arpa's *gate*), that makes calls to the Internet on behalf of an inside machine and relays bytes between the inside Datakit connection and the outside Internet TCP connection. *Proxy* can also listen to a non-privileged socket and report connections to an inside process. Several outbound services are implemented using *proxy*, and more are easy to create. In all cases, it appears to the remote Internet hosts that our gateway machine is making the calls.

Inet may be reached over the Datakit. But how do internal machines reach inet over the Ethernet? R70 responds to two IP addresses: its own, and an internal IP address for inet. (Dave Presotto implemented this after a trivial change to the Tenth Edition Research Unix connection server.[connection]) Calls to certain TCP ports on this internal IP address invoke *dcon*, a program that simply relays the bytes between the TCP port and Datakit connections on inet.

I have replaced the old *aftp* and *atelnet* with *ptelnet* and *pftp*. They work in the same manner, but the new routines call a portable implementation of *ipcopen*, a piece of the connection server. *Ipcopen* hides the details of a connection (TCP sockets or Datakit), simplifying the application program. For example:

```
ptelnet tcp!toucan
```

connects to machine toucan on our internet, and

```
ptelnet proxy!ernie.berkeley.edu
```

connects to ernie.berkeley.edu on the external Internet. proxy! is the default. The *ipcopen* implementation is not flawless: some socket features such as out-of-band data and the urgent pointer are missing because they are not supported by Datakit. *Ptelnet* was stripped down to avoid these features.

Pftp provides FTP access in a similar manner. It is an updated version of *aftp* from arpa. The *ipcopen* routines allow it to work over Datakit.

Outgoing mail is sent to inet via *smtp* over either Datakit or the internal Internet. It is stored and forwarded from there. Upas performs the mail gateway functions.

Inbound services

We provide incoming login and mail service. For incoming file transfer, inet provides an anonymous FTP service.

We do not trust our passwords to the Internet: it is too easy to eavesdrop or steal packets. See [smb] for a discussion of these security problems. Login service requires a hand-held authenticator (HHA). These are calculator-sized devices that contain DES encryption and a manually-loaded 64-bit key. They cost about $50.

Inbound login service is provided through an authentication manager on r70. A session is shown below:

```
$ telnet research.att.com
Trying...
Connected to research.att.com.
Escape character is '^]'.

RISC/os (inet)
login: guard
RISC/os (UMIPS) 4.0 inet
Copyright 1986, MIPS Computer Systems
All Rights Reserved
Security Authentication check
login: ches
Enter response code for 90902479: 818b71fe

Destination please: coma
OKYou have mail.
coma=; date
Tue Nov 14 10:52:37 EST 1989
coma=;
Eof
Connection closed by foreign host.
$
```

To connect, the following sequence occurs:

- The Internet caller uses *telnet* to connect to research.att.com (a.k.a inet) via *telnet*. The login name is guard.
- The guard login connects to the authentication manager on r70 over the Datakit. It spends the rest of the connection relaying bytes between the two connections.
- The authentication manager on r70 requests a login name.
- R70 sends a random challenge number, which the caller supplies.
- The user enters the challenge into his HHA.
- The HHA encrypts the challenge using a preloaded DES key, and displays the response.
- The user types the response. He has three tries to answer a challenge correctly, and is disconnected if he fails.
- The authorization manager prompts for a Datakit destination.
- When the user enters the destination, the manager sends a redial request to the Datakit controller with the given destination and a service of 'dcon'. For machines that trust r70, the 'dcon' service bypasses further logins and avoids further passwords.
- The redial request transfers the call, switching r70 out of the connection. In non-Datakit implementations, r70 would probably have shuttle bytes between the two connections.

Each user requires a DES key, and keys have an expiration date. The keys are stored on a separate passwd/key server machine connected to r70. The keys in this machine may be changed or examined only from its console.

Inbound mail is delivered directly to inet. Inet checks the destination. If it is a trusted machine (i.e. its *smtp* is trusted), a connection request is sent to r70. If not, the mail is relayed through an accessible internal machine. R70 will permit connections only to trusted *smtp* implementations. The list is short because most internal machines run *sendmail.*

Protecting INET

The preceding precautions might imply that we expect our gateway to be compromised at some point. In fact, we are taking great pains to protect the machine, including the usual good system administration steps needed to secure any Unix system[ritchie]: directory and file permissions are checked, backups performed regularly, etc.

We have taken some steps to avoid denial-of-service attacks. For example, the logs, the spool directory, and the publically-accessible FTP directory are each on separate file systems. If a stranger fills the public FTP directory, there is still room for the logs.

Here are some other steps taken:

- All the important executable files are periodically checksummed and checked for changes.
- Most user accounts do not have passwords to be checked. They obtain permission to login based on the source of the call.
- Non-essential network daemons have been removed: we don't need to trust them.
- *Inetd(8)* handles all network connections. Certain modifications allow *telnetd*, *smtpd*, and *ftpd* to run without special permissions:[ritchie] *inetd* handles the privileged stuff.
- There is extensive logging of network activity, including connection and login attempts. A write-only log server is planned that will keep a copy of these logs off-machine and inaccessible to any network.
- Since the network daemons are so important to the security of the machine, we obtained the latest BSD versions and examined, modified, and installed them.

Gateway alternatives

There are several much simpler alternatives for an Internet gateway. The simplest is a router, which just lets the packets through. Some routers, like

Cisco's, provide packet filtering that can block various types of access to an institution.

We did not choose the router. Though the filtering is quite good, it's not clear whether a clever worm could get through the permitted ports. Can we trust the router? If *telnet* access is allowed from the outside, inside machines are exposed to password-guessing attacks. If *telnet* access is not allowed, an alternative is needed anyway, requiring additional provisions. The router does not provide logging to detect invasion attempts. And mail gating must be provided by a machine somewhere: it is unreasonable to expect each internal machine to be configured to handle all the varieties of external mail addressing.

Many Internet sites use a gateway machine like a Sun. These machines forward IP packets in both directions, and provide a mail gateway service. The packet flow is still dangerous, though filtering is available. Many internal machines may trust the gate machine, leaving them further exposed if the gate machine is compromised.

Performance

The mail throughput of the new gateway has been gratifying, though a VAX 750 is an easy act to follow. In many cases, we have had replies to cross-country mail return in less than a minute. It sometimes seems that the mail must have bounced. Inet has little else to do, and a MIPS M/120 is a fast machine.

Pftp transfers are fastest over Datakit, since they avoid the *dcon* gateway in r70. File transfers range from 17 to 44 Kb/sec. TCP transfers through r70 run at 9 to 16 Kb/sec. By comparison, *ftp* on inet runs at about 60–90 Kb/sec. Clearly, security has its costs. But these are top speeds. The limiting factor is often the external net or host. In any case, several users have expressed satisfaction about the throughput.

Conclusions

The new gateway achieves a useful balance of utility and security. Most internal users seem to be happy with *pftp* and *ptelnet*. Some have asked for *talk*, resolver service and other UDP-based protocols. These could be provided with non-*proxy* services on inet accessible through Datakit.

There are certainly limits to our security. If r70 and inet are subverted, the inside machines could be attacked.

Insiders can easily import trouble such as Trojan horses or programs infected with viruses. Our best defense is continued scanning of internal machines for security holes in case such a program gets loose.

There is now a second AT&T internet gateway. Its configuration is similar to inet's. These two front doors provide reasonable security to an isolated internal internet. But AT&T is a large company, so we keep a constant watch to assure that no other links are made to the external Internet. A locked front door is useless if the back wall of the house is missing.

The incoming guarded *telnet* service is not perfect. The remote *telnet* may be insecure, and the TCP connection itself could be stolen after login is complete. Most internal AT&T machines do not accept r70's judgement that the user is valid, and require their own login passwords. These passwords travel over the Internet in the clear.

Our solution does have some drawbacks. We rely on two machines and Datakit to keep things working. This yields three points of failure, while the simpler approaches have (in some sense) only one point of failure. The use of TCP-level gateways does lower throughput. Though most users seem to be content with the *pftp* response, it would be nice to speed it up some.

**This paper is not an invitation to come test the security of our gateway.
It is management's policy to call the authorities when intruders are detected.**

Acknowledgements

Many people have contributed to the support of these gateways. Steve Bellovin did most of the initial work to get arpa talking to the ARPAnet and Datakit. Dave Presotto has supplied much of the software and most of the paranoia to provide a secure gateway. Howard Trickey implemented earlier versions of *ptelnet* and *pftp*. Dennis Ritchie has kept a watchful eye and stepped in when things broke. Steve Bellovin and others have provided numerous suggestions and warnings on various networking and security topics. Jim McKie ported many useful Research Unix[V10] functions and the INCON Datakit driver to our MIPS computers, making life much easier for me.

References

[1] David Presotto. Upas - a simpler approach to network mail. USENIX Summer Conference Proceedings, pps.~533–538, June 1985.

[2] Donn Seeley. A Tour of the Worm. USENIX Winter Conference Proceedings, Jan. 1989.

[3] David Presotto and Dennis Ritchie. Interprocess Communication in the Ninth Edition UNIX System. Unix Programmer's Manual, Tenth Edition. A. G. Hume and M. D. McIlroy, Editors. AT&T Bell Laboratories, Murray Hill, NJ. 1990.

[4] Bellovin, S.M. Security Problems in the TCP/IP Protocol Suite. Computer Communications Review, Vol. 9, No. 2; April, 1989, pps. 32–48.

[5] Dennis M. Ritchie. On the Security of UNIX. Unix Programmer's Manual, Tenth Edition. A. G. Hume and M. D. McIlroy, Editors. AT&T Bell Laboratories, Murray Hill, NJ. 1990.

[6] Unix Programmer's Manual, Tenth Edition, Volumes One and Two. A. G. Hume and M. D. McIlroy, Editors. AT&T Bell Laboratories, Murray Hill, NJ. 1990.

USENIX Summer Conference
June 11-15, 1990
Anaheim, California

Putting UNIX on Very Fast Computers

Mike O'Dell – Bell Communications Research

ABSTRACT

A computer with a 250 MHz clock and built from leading-edge technology works in fundamentally different ways compared with a one-chip CMOS VLSI processor clocking at less than 50 MHz. The interactions between a UNIX implementation and its supporting hardware have always been quite subtle and remain a considerable headache for those charged with porting the system. But in addition to the imprecisions of fuzzy functional definitions for some key system facilities, the laws of physics conspire to make the marriage of very fast computers and modern UNIX systems an even more interesting challenge than it would normally be. This paper discusses some of the matchmaking necessary to achieve a matrimonious accommodation.

Introduction

Computers which operate on the hairy edge of physics are radically different beasts than the bus-based minicomputers where UNIX was born. Many machines of this latter class are now called "workstations" or "servers." Traditional supercomputers are not ideal platforms for UNIX, given their penchant for word addressing and limited memory management hardware. For many years, though, they were the fastest game in town by a large margin, so the architectural inconveniences in the name of raw speed were simply endured. Recently, though, this has started to change. High-performance implementations of what one might call the new "commodity instruction sets" hold the promise of providing supercomputer-class scalar performance with all the amenities that modern UNIX systems want to see in the underlying hardware (support for demand-paged virtual memory, for example).

The modern reality is that, rightly or wrongly, the software base available for a machine goes a long way toward determining its success in the marketplace, so building a new computer with Yet Another Instruction Set, and therefore no existing software base, is an undertaking fraught with enormous risks. To the reasonably risk-averse person (e.g., the people who fund such projects), it is clearly advantageous to *leverage*[*] the large installed software base associated with an established instruction set and operating system implementation. (The IBM PC clone-builders are some of the most successful proponents of this approach.) Going even further, if the new machine is sufficiently like its archetype, one can even leverage a very large part of the base operating system software implementation. This approach has the potential of maximizing return on investment while minimizing risks. The overall implication is a very high degree of compatibility even at the level seen by the kernel, with perfect compatibility when seen by user-mode programs. As was discovered, however, this "perfect user mode compatibility" can be a rather elusive goal when the differences between the underlying implementations are dramatic enough.

There are two ways to pursue the goal of building a mainframe-class, very fast instruction-set clone. The first is to use the highest performance off-the-shelf processor chipset available as the core of the machine and then concentrate on the rest of the computer to add value. This is a potentially workable approach because the instruction pipeline is only a small part of a large computer – the memory and I/O systems of a mainframe-class machine can easily consume the lion's share of the logic packages and the engineering effort needed to build the whole machine, particularly if the processor is but a few VLSI chips. The big advantage here is that many of the subtle instruction-set compatibility issues are resolved by simply buying a working processor chip. One problem with this approach, however, is that such a machine probably can't go dramatically faster than any other machine built from the same, commonly available chips. If one intends for performance to be the fundamental, market-distinguishing characteristic of a new machine, this may not be the best approach.

The other alternative is to license a machine architecture from one of the several purveyors, and then set out to build a new implementation which squeezes every iota of performance from whatever base technology one feels comfortable choosing. In this approach, the down side is that you must now build *all* of the computer; the up side which may make the approach worth the risk is that one now

[*] Boardroom talk for "use", "run", or "execute unmodified."

gets a *chance* to achieve a real performance margin over machines built with commodity processor parts. At Prisma Computers, Inc., we embarked on this second path – construction of a supercomputer which would be binary compatible with a popular workstation family, and built from "whole cloth." For the purpose of this effort, the definition of *binary compatible* was beguilingly simple: if a binary program runs on one of the to-be-cloned workstations, it must run·identically on the new machine, only much, much faster.

In the course of designing this computer and preparing an operating system for it, we discovered many strange and often confounding things. Most of these issues relate directly to what promises a machine and its operating system must make to programs. Particularly troubling are the dark corners or boundary conditions in an architecture which ultimately are revealed to a user program in great detail. Programs often have deeply-ingrained notions about what they expect to see in these revelations, and what they have come to expect of certain hitherto inadequately-specified operating system features. These features can be viewed as an example of "inadequate law," so as with traditional jurisprudence, one must fall back on the available historical precedent which is seldom more compelling than the statement, "This feature has always worked on other machines, so my program is free to use it!" Programs which take this view of the world make "absolute binary compatibility" an interesting concept and a tricky goal to achieve.

At this point, it is useful to examine a new player in the compatibility game: the Application Binary Interface, or ABI, which was developed for UNIX System V Release 4. To understand the importance of the ABI, we need to look at history for a moment. One of the great success factors in the IBM PC world has been the large software base available for the machines. Historically, this has been achieved by everyone simply building the same computer (as seen by programs). In the PC world, the definition of "compatible" has been: if a program or operating system works on a genuine IBM Personal Computer of some flavor, it should work on a Brand-Y PC clone; if not, the Brand-Y machine is broken. The problem is that in the UNIX world, the design space is much, much larger than the PC world, so expecting everyone to build similar machines isn't reasonable. However, if UNIX is to take off, it is vitally important that all UNIX systems using the same instruction set be able to execute the same binaries so vendors can market "shrink-wrapped" user application software. The ABI, then, is an attempt to provide the needed compatibility at the user level while not overly constraining the operating system implementation.

An ABI is created for a specific architecture and defines *all* system interfaces available to user-level programs on any implementation of System V Release 4 running on that architecture. The operating system's internal structure, on the other hand, is allowed to be changed as desired (e.g., for efficiency or performance), as long the changes don't violate the ABI specification seen by user-mode ABI-compliant programs. The goal, then, is that ABI-compliant binaries for a given architecture should run on all implementations of that architecture, with which provide the constant, basic functionality specified by the ABI. In general, the ABI specifications try to be as high-level as possible, striving to avoid documenting details which should ideally be left as implementation choices. However, a great many things must be specified if programs are to achieve their portability goals so some of the problems addressed later in this paper are still real issues. The reason that Prisma didn't simply provide an ABI-compliant interface, instead of the more rigorous PC-style absolute user compatibility, was simply one of timing. The operating system for the Prisma P1 was not going to be System V Release 4 until well after the first shipments; and a very critical business issue was leveraging the existing large third-party software catalog without any changes. Some day, ABI compliance will be sufficient for a new machine, but it wasn't in Prisma's development time frame.

The Rules of the Game

The Prisma P1 was originally designed to a target clock cycle of 4 nanoseconds, and since the P1 was a RISC machine, the fervent hope was that it execute very nearly one instruction per clock. To give you a feel for this target clock rate, a 4 nanosecond clock period translates to a 250 MHz clock frequency (well into the UHF television broadcasting band). At these frequencies, signals are *not* digital. It is quite true that the intent is that the signals represent ones and zeroes, but at these frequencies, the signals themselves are profoundly analog.

If the machine is to run with a 4 nanosecond clock, the logic used in the machine must have loaded gate delays of about a hundred picoseconds, with the rise times of a 250 MHz "square wave" logic signal measured in a few tens of picoseconds. The harmonics of such signals extend well above 2 GHz into the serious microwave and radar frequencies. This means that "wires" or "traces" on a circuit board are all transmission lines, and keeping the signals contained in the desired paths becomes an RF engineering task. Because the signal traces are transmission lines, the trace routing is further complicated by the restriction there can be no "stub" paths branching off a main run to a non-collinear connection on the way. Instead, the signals must be routed in a purely linearizable "connect-dotswithout-lifting-your-pencil'' path. Complicating matters even further, the transmission line traces were ''differential pairs,'' which means that instead of running one wire from place to place, Prisma ran two, with one wire going from one to zero while the other goes from zero to one at the same time. The logic signal is taken as the difference between the two lines (hence the name ''differential'') instead of between a single line and ground as in the case of standard TTL logic. This technique has important electrical propagation characteristics as well as some handy logical advantages – for instance, inverters are almost unheard of – just reverse the connections! These traces must be routed on the circuit board maintaining a specific distance between them in order to preserve the characteristic impedance, while being routed as a pair introducing only an *even* number of half-twists en route to maintain signal polarity. PC board routing software which can adequately do this complex job proved quite hard to come by.

On real circuit board material manufacturable at finite cost, electronic signals propagate noticeably slower than the speed of light in a vacuum. The rule of thumb for the board materials used in the P1 was that 1 inch of circuit board trace eats 180 picoseconds of flight time, so 10 inches of trace will eat about half the available clock cycle. Given the differential-pair stub-free routing requirements, it was frighteningly easy to get routes on a 4 inch by 4 inch board which came dangerously close to 10 inches, thereby leaving the logic in such a path about 2 nanoseconds in which to do all its work.

All of these signalling and routing requirements also apply to connectors and cables. The most vexing issues are that signals generally propagate even slower through cables, making impedance control for minimize reflections a constant concern. These issues particularly complicate the physical construction of tiny, fragile connectors which require many, many pins to interconnect modules with ever-growing complexity.

One of the most serious problems Prisma faced was the level of integration available in the Gallium Arsenide (GaAs) technology being used for the machine. The GaAs logic chips were medium- to large-scale integration, meaning that a chip contained between about 200 and 500 gates, as opposed to the tens-of-thousands of gates available on a CMOS VLSI chip. If a function needed more gates than would fit on a chip, the signal path had to go off-chip, paying dearly for driving the signals across the circuit board instead of across the chip surface. This made the partitioning and physical floor plan of the logic and the resulting chips quite critical. There were never enough gates, so only functions which were absolutely essential for correct behavior could be done in the GaAs logic. The processor had no choice but to be a RISC – there wasn't enough real estate to allow alternate ideas.

One other characteristic of the interconnection constraints had a profound architectural impact. At the speeds Prisma was targeting, there is no such thing as an arbitrated ''bus.'' The data interconnects in the P1 were nominally 64-bit wide, 4ns clock, simplex point-to-point data paths. An ''address path'' was just a data path which carried address information as the data and usually required fewer bits than the 64-bit data paths. A full-duplex data path (two simplex paths which connect the same two parts of the machine but in opposite directions) required about 300 wires total (the signals are all differential, so 64 bits of data require 128 wires each direction; differential parity and control signals consumed the rest). These paths were hideously expensive in both space and time, so adding even one path to the implementation required an overwhelming argument in its favor – on the order of ''The machine cannot possibly get the right answer if it's not there.'' Because there were no busses carrying a combined data and address path, there was no convenient place to put useful trinkets like EPROMs or UARTs which have come to be expected for mundane tasks like booting and printing last-gasp death messages when something goes horribly wrong. This constraint, more than any other, forced the machine to have a service processor. (The kernel development group successfully avoided having anything to do with the service processor beyond defining some requirements and a few necessary interfaces needed between the ersatz ''ROM'' code placed into P1 memory by the service processor and the kernel.)

The final, most important rule of the game was that the machine had to be realizable using parts which are actually available – not vaporspecs for non-existent chips, but *real* DRAMs and *real* SRAMs and *real* logic chips (or at least a real fabrication process), running at speed, which one could buy (or make) in production quantities, at rational prices. This constraint is a genuine nuisance.

All the rules from physics and engineering discussed above induced two fundamental constraints which profoundly effected everything in the design of the Prisma P1:

(1) Doing anything takes at least a clock, and

(2) Getting there to do it often takes at least a clock as well.

This means that the back end of the instruction pipe *cannot* communicate with the front of the pipe in one clock cycle, and therefore, there can be no ''atomic'' decisions whereby the instruction fetch unit decides what to do next based on the entire state of the CPU (e.g., take a pagefault). This presents a non-trivial problem.

Suggestions on How to Play the Game

The general problem to be addressed is that processing an instruction always takes too long, either because the logic takes too long to get the answer, or that the logic to be consulted is too far away as well as being too slow. One of the most productive ways to hide latency is to increase parallelism, thereby letting more operations progress at once. This means that functions which are one clock on other machines became pipelines on the P1 and therefore had to cope with multiple outstanding operations.

The biggest latency issue was the dramatic disparity between the speeds of the memory system and the processor. Since the machine was to be relatively low cost and support large memories, hidden somewhere behind the curtain had to be commodity 80 nanosecond DRAMS. Note well that the 80 nanosecond number is *access* time, not the transaction time, which is on the order of 180-200 ns. This limit drove the decision to make the memory system a 3-level hierarchy. The first level instruction and data caches were integral to the GaAs portion of CPU. It was indeed a dark day when it became apparent that the GaAs RAMs we had planned to use in the primary caches were not real, and that high-performance ECL "474" RAM parts would have to be used instead. This meant that the primary caches would have to become pipelined: the first 4 nanosecond cycle did the cache tag access, and the second cycle accessed the cache line and returned data. The primary caches could start an operation every clock and the data cache would have to support multiple outstanding cache misses. To do this for both loads and stores, the primary data cache implemented the equivalent of load-store locks for all of main memory. The second level cache was a very large, interleaved, integrated physical cache front-ending the third-level DRAM arrays. I/O traffic went through the secondary cache with some clever tricks to prevent cache-wiping during I/O bursts. For a load, a hit in the secondary cache returned data to the CPU register in about 15 cycles round-trip (updating the primary data cache en route), while awakening the DRAMs from their deep slumber required 60+ cycles round-trip.

As was intimated in the description of the primary data cache, load and store instructions were implemented as non-blocking operations. For example, a primary data cache miss caused by a load did not stall the pipe until a later instruction attempted to use the destination register of the load as a source operand. Because the cache miss could be answered by either the secondary cache or the DRAM array, load instructions could complete out of order. When coupled with the non-atomic behavior of the pipeline, this could cause multiple outstanding pagefaults to occur, and resulted in considerable grief for the operating system. While the non-blocking loads and stores added significantly to the complexity of both the hardware and software, the system simulator clearly showed that this feature had a dramatic impact on machine performance and that the work in the kernel was well worth the effort.

Another area impacted by the speed disparity between the processor and memory system was the architecture of the memory management unit (MMU)– both in the page table structure and the translation look-aside-buffer (TLB). In some circles, with processors operating at more modest performance levels, it is considered reasonable to implement an MMU with only a software-reloaded TLB; but for a machine like the P1, such a design is untenable for performance reasons. The P1 was designed to be a supercomputer and to run processes requiring large address spaces, so the TLB-miss processing speed, directly related to the average page-table walk length, was an important issue. The "reference MMU" for the P1's underlying RISC processor uses a short-circuit 3-level page table with 4Kbyte leaf pages. The problem with this design (from the P1 perspective) is that the deep page table tree required a large and rather clever TLB design to prevent multiple memory references when walking the page tables for a TLB miss. While CMOS implementations of the reference MMU easily have enough transistors to support the architecture, the required cleverness was beyond the means of the GaAs implementation technology used in the P1. Simulations studies of alternative designs led back to a two-level version of the reference MMU using 8Kbyte pages. This approach allowed the TLB to be divided in half, dedicating half to level-one entries and half to level-two entries. The resulting design allowed processes with very large address spaces to exhibit very low TLB miss rates, and when a miss did occur, the mean page-table walk length was well under one entry. The entire MMU design effort was a case where the synergy of developing the kernel and the hardware in parallel proved extremely valuable. Several "strong candidate" designs were actually coded into the architectural simulator being used for kernel development and were evaluated by running programs (including the operating system) before the final design was frozen.

In summary, building a machine as fast as we intended the P1 to be requires all the tricks you can think of, and probably more. It also requires a great deal of attention to matters which are second-order effects (if not third!) on slower machines. In fact, you almost end up standing on your head; the first-order and other-order issues almost seem to be reversed in this world - things which matter a lot in slower or more cost-sensitive designs are often down in the noise, and things which most designs don't attempt to do anything about matter a great deal.

Great Expectations – What UNIX Wants from Hardware

UNIX systems at least as early as the Sixth Edition used their hardware in interesting and aggressive ways. The technique used by the V6 system to grow the stack segment on demand assumed instruction restart capability which was easy to do on the DEC PDP-11/45 and PDP-11/70 because of the splendid support in the MMU. Providing the equivalent capability on other members of the PDP-11 product line was considerably less trivial. The early microprocessor-based UNIX implementations (V7 from UNISOFT on the Codata 3200, as an example) had to deal with the unavailability of instruction restart on Motorola 68000 processors. Various tricks were devised, but the general approach was to add a ''sacrificial instruction'' to the procedure prolog code which did a probe of the address space designed to provoke a stack-growing protection trap if required. The instruction could not be salvaged, so the MMU trap recovery code looked at the offending instruction to see if it had the designated sacrificial form and seemed to be just probing the stack. If so, then the instruction was just ignored. This is the first example known to the author where a restricted but unprivileged instruction sequence was used to paper over an inadequacy of the underlying hardware behavior. Other UNIX implementations have used similar work-arounds to deal with various problems, often relating to MMU and trap recovery behavior, but the days of those tricks covering for inadequate hardware support are over.

Modern UNIX systems such as System V Release 4 demand much more from the underlying hardware. While one might argue academically that copy-on-write is not strictly required for a System V Release 4 implementation, it is most certainly required if the implementation is not to be at a serious disadvantage in the real marketplace. Furthermore, support for facilities such as copy-on-write must not exact excessive performance penalties.

Adding to the complexity of providing these facilities at high speed are new processor architectures which specify what we call ''atomic trap observance.'' Atomic traps appear to happen as part of the instruction execution. For example, a load instruction which pagefaults is specified as causing the fault, rather than completing the load, as part of the instruction execution. The machine must begin the trap processing instead of executing the next instruction after a pagefault.

The inclusion of this atomic behavior in the architecture is understandable, particularly if you have ever worked on fault recovery code on a complex machine. The beguilingly simple specification is a blessing for the low-level kernel programmer because it makes fault recovery quite straightforward and generally makes instruction restart quite easy to achieve. However, if the MMU which detects the pagefault is 3 or 4 pipeline stages removed from the instruction fetch unit, and when, because of out-of-order completion, at least two of those pipeline stages can irrevocably alter registers which were used to form the effective address of the load or store, the operating system is faced with several serious problems.

The first problem is simply assuring the fundamental semantic correctness of operations such as copy-on-write by ''re-effecting'' out-of-order instructions (since one cannot just backup the program counter!). The approach taken in the P1 was to replace the usual ''MMU Registers'' which record the necessary trap recovery information with a pagefault queue whose entries provide enough information to ''re-effect'' the faulting instruction in spite of its out-of-order completion.

For the private purposes of the operating system kernel, the MMU Fault Queue provided all the support needed to recover from pagefaults and continue, and it worked quite well on the simulator. The simulator used by the kernel group actually allowed more asynchrony than the real P1, thereby catching a few subtle bugs which would have been much harder to find running ''at speed'' on a real P1. The real nightmare, however, related to facilities afforded to User Programs.

Modern UNIX systems (or possibly more aptly, systems gene-spliced with TENEX) now have virtual memory systems which let user programs actively participate in memory management functions by allowing them to explicitly manipulate their memory mappings. This capability is not a problem in and of itself, but rather, serves as the courier of an engraved invitation to Hell. In this Brave New World, programs may register signal handlers for pagefaults and in turn, they expect to take the fault, diddle some mapping information, and return from the fault to carry on as if nothing happened. All the clever design work to make the P1 go fast and still let the operating system transparently make things come out right seemed to be torpedoed by this one requirement. Several loud voices suggested that such programs should just get what they so richly deserved, but when it was pointed out that the Bourne shell was one of the most commonly-executed offenders*, that alternative was clearly unacceptable. The stated business requirements for absolute binary compatibility were also persuasive. Even more unacceptable, however, was slowing down the entire machine by a large factor just to support infrequent exception handling.

*The Bourne shell uses *sbrk()* to manipulate the mappings instead of *mmap()*, but it does catch the appropriate signal and expects to return and restart the failing instruction nonetheless.

Interestingly enough, most architectures already have "non-atomic" or deferred traps generated by long-running coprocessor instructions (usually floating-point), and programs seem to have resigned themselves to dealing with them as a matter of course. An interesting case to consider is what must be done in a language with frame-based exception handling (Ada or Modula-3) to deal with returning from a subroutine before all the floating-point instructions issued in the subroutine have finished. The unfinished instructions are still subject to the exception-handling wrapper around the invoking subroutine call. Therefore, an explicit instruction to synchronize the state of the floating-point coprocessor must be used in the procedure return sequence to prevent the integer pipeline sequence from completing too soon (thereby abolishing the exception-frame environment while incomplete coprocessor instructions could still cause exceptions). This need to be explicitly aware of the potential asynchrony has resulted in lowered expectations as to exactly what information can be revealed about the exact state of the machine when the trap is induced, and lowered expectations about what a program can do as a result of a trap (delivered to the user program as a UNIX signal), at least for coprocessor traps. The problem with atomic trap behavior for loads and stores is that it essentially requires synchronizing the entire memory system pipe on *each* load or store instruction, and this completely defeats all the effort to pipeline those functions in order minimize latency by maximizing overlap. It is unfortunate that the direction processor architectures in general seem to be headed is toward more synchrony and lock-step determinism in such behaviors when it can be readily shown that reducing synchrony and determinism is a powerful way to improve performance.

Another expectation that UNIX systems have about machines has to do with the way kernel and user address spaces co-exist. The UNIX system at issue clearly wanted the kernel to be mapped into the upper one-half gigabyte of the user's 4 gigabyte virtual address space, leaving 3.5 gigabytes for user programs. Indeed, the System V Release 4 ABI for the P1's underlying RISC architecture requires that the valid user virtual address space be 3.5 gigabytes starting at location zero, so changing the virtual memory layout in a substantial way was never a realistic option. Since the primary caches use virtual address tags for reasons of speed, redundantly mapping the kernel into each user address space caused the *entire* kernel to alias in every address space. Since the primary cache also had process-id context bits in the tags to prevent unnecessary flushing and cache-wiping, a naive implementation would require flushes on every context switch to insure kernel coherency. The first alternative (discarded rather quickly) was simply marking the entire kernel "no-cache" in the page tables, but the performance impact was too severe. The alternative chosen was to modify the MMU behavior equations so that the uppermost .5 gigabytes of virtual space always got processed as if in context zero, with the proper considerations given to the supervisor or user mode of the processor. This prevented aliasing by forcing the kernel to only appear in one context.

The advantage of this design is that it allowed aggressive cache-flush prevention algorithms which were added to the virtual memory system to perform quite effectively. The disadvantage is that to some degree, the decision permanently forces a considerable chunk of user virtual space to be dedicated to kernel use, whether or not it is needed down the road. The implications of not making these decisions, however, were considered to be much worse, based both on calculations and tests validating the effectiveness of the cache-flush prevention algorithms.

Great Assumptions – What Programs Think UNIX Provides

The complexity of the UNIX virtual machine assumed by programs (and only partially documented by the system call interface manual pages) only seems to be increasing. In the halcyon days of the Sixth Edition, the manual page for the *signal()* system call warns that signals were primarily a way for a suddenly-dying program to discover its assailant, and not the event-based interprocess communications mechanism they have become of late. Admittedly this author would be the last person to complain about signals becoming reliable, but rather, the complaint is that the flexibility of the interface has not tracked the richness of use. Again, the issue of user programs intercepting pagefault traps with a signal handler was at the center of the problem.

In order to resolve the problem of the significant performance impact caused by the architecture specifying atomic trap behavior for pagefaults, a cooperative effort between Prisma and the purveyor of the architecture produced a revision to the architectural specification which provided that *deferred* traps, already legal for floating point operations, could also be legally generated by certain additional kinds of exceptions (e.g., pagefaults), but *only* when a User Program has not requested visibility of the trap. On the face of it, this sounds like a reasonable compromise. The system is allowed to do whatever it wants, ripping along as fast as possible, as long as no one asks to see the details. On the other hand, if a process asks to watch the entrails in action, the machine must behave in such a way as to allow the operating system to reveal to the interested User Program all the relevant gore about what was happening, and in such a way that the user program can diddle with the state and have the diddling take immediate effect. And how does a program request to view the entrails? By registering a signal handler, of course.

This still sounds just fine – programs that ask get hurt; those that don't, don't. There is still a serious flaw with in model, however. A great many programs (all programs linked with the FORTRAN run-time system, to pick just one group at random) routinely catch *all* signals in an attempt to print the moral equivalent of:

```
yourprogram bailing out near line 1
      with Segmentation Violation
```

all in the name of "user-friendliness." The program isn't going to try and recover from the disaster; it only wants to report the fatal mistake and then die peaceably. By registering the signal handler for the appropriate signal, however, it incurs all the penalties of a program expecting to examine every trap while trying to do its own demand paging. As was said above, in the case of the Prisma P1, these penalties have a quite serious performance impact. The crux of the problem is that a user of the *signal()* system call doesn't have any way to request a specific "quality of service" – so the system must provide the most expensive, guaranteed-to-be-sufficient service, when the barest minimum would be perfectly adequate in many cases.

Great Consternations – Life at the Boundary Conditions

No UNIX Programmer's Manual known to the author contains detailed information about what questions a signal handler can expect to ask of the machine; about what state information must be revealed in response to those questions; nor about how a signal handler can expect to effect the state of the running program, particularly by modifying the machine state revealed to the signal handler and then "returning" from the signal handler. Indeed, in some programs, the signal handler doesn't ever return – it does a *longjmp()* to forcibly redirect the control flow of the program! The ANSI C standard attempts some limited statement of allowable behavior applicable in the broadest, most portable cases which must work under essentially any operating system, but it is insufficient. The current state of confusion exists because under specific operating systems, certain behaviors have been discovered to work and they have largely been perpetuated, even if undocumented. One solution is to categorize "quality of service" for signal handlers, and thereby allowing a handler to register its intent and its requirements as to the completeness of information needed by the handler. In this way, a program need not pay in reduced performance for more information than it needs.

Prisma implemented a very limited (lame?) version of this quality-of-service notion by providing a new system call named *sigdefer()* for manipulating a mask which indicates that certain traps should remain deferred even if the program registers a signal handler for them. This mask is potentially "sticky" across *exec()* system calls, thereby providing support for a **sigdefer** command which takes another command as its argument. The **sigdefer** command sets the "force deferred signal" mask for the indicated signals and then runs the command, thereby allowing it to run unimpeded by *signal()* calls intended to only print messages. While knowledgeable programs can use the new system call directly, the **sigdefer** command allows existing binaries to run at full speed when the user decides it is safe. Within the kernel group, the **sigdefer** command was called, at various times, **gofast** and **benchmark**. This is approach is admittedly a bit disreputable, but it solved the problem until someone defines a quality of service interface for signal handlers.

In return for specifying some behavior more flexibly, this author would like to encourage the use of the word *undefined* more frequently in standards and architectural behavior specifications. While *defined* means that a certain behavior is guaranteed, the author intends *undefined* to mean:

> Even if you can somehow discover how an *undefined* facility behaves on an implementation, you are expressly prohibited from using that knowledge in any program because any implementation is completely free to give you a different answer EVERY time you use it!

Maybe the threat is enough to make people stick to the *defined* behavior, but it is delicious to contemplate making an *undefined* behavior actually work differently each time!

Comments and Conclusions

Building fast computers is an interesting and exciting challenge, particularly when you first discover that *Everything You Know Is Wrong!* In doing the kernel for the P1, the biggest single cause of grief was not the complexity arising from nominal behavior of the machine (even including multiple outstanding pagefaults), but rather it was because programs, especially in signal handlers, are allowed to ask arbitrary questions and request arbitrary behavior of the machine at very inopportune moments. Given the current operating system interfaces (ABI specifications), answering the questions and implementing the requested behavior can only be done at the expense of serious performance penalties. These penalties can unfortunately be inflicted upon unsuspecting programs just trying to be friendly with the user, rather than only on those programs truly interested in exactly what the machine is doing on a cycle-by-cycle basis.

Historically, UNIX programs have taken a quite naive view of the world: system calls always work (who checks error returns?), *malloc()* never fails, and a signal handler is free to crack stack-frames and diddle saved register values to its heart's content with perfectly predictable (or at least, experimentally discoverable on a specific machine) results. Well, maybe you should check the return value from *unlink()*; some machines rap your knuckles quite smartly for dereferencing through Location Zero; just maybe catching that signal is a lot more expensive than you thought; and most certainly, meddling with the saved state to achieve a particular effect is much harder than you thought. In the future, machines will continue to get faster and more internally concurrent, if not more parallel, and continuing this fantasy of the world being atomically lock-step-perfect simply flies in the face of the physical reality of fast machines. The sooner we can disabuse programs and ABI specifications from this fantasy, the sooner we can really exploit of the machines we could build.

The kernel for the Prisma P1 contained a lot of very creative work; this paper has described only a small part of the issues we faced in trying to make a workstation-tuned operating system safe for a serious supercomputer. The virtual memory system, the filesystem, the scheduler (see [Essick]), the basic I/O system and fundamental system resource management were all modified in various ways.

One final note: The kernel for the Prisma P1 was up and running in two forms. A complete P1 kernel except for the lowest-level machine-specific MMU hardware support and I/O system code ran on a production basis on all the large compute servers at Prisma, day-in and day-out. This system contained over 95% of the code in the ready-for-hardware P1 kernel. The entire P1 kernel, complete with P1-specific MMU and I/O system code, was running solidly on the P1 Architectural Simulator under extreme stress-test loads.

Acknowledgements

The author did not learn any of these things alone, rather they were learned in a very creative and rewarding group effort. Brian Berliner, Ray Essick, Bill Fuller, Tony Schene, and the author comprised the Prisma P1 Kernel Team. Bill Wallace made the Prisma software group a great place to work. Chris "Roz" Rozewski and Rob Ober created the MMU and the memory system for the Prisma P1, and Mike Baird's group designed the I/O system. The author is very grateful to all of them for their contributions and the chance to work with them.

References

[Essick] An Event-based Fair Share Scheduler, Raymond B. Essick, January 1990 USENIX Proceedings, Washington DC, USENIX Association, Berkeley, CA.

Mike O'Dell recently joined Bell Communications Research after having served as Chief Computer Science at Prisma Computers. Prisma's goal was the construction of a very high speed GaAs SPARC compatible computer with fast memory and I/O buses. He is Vice-President of the USENIX association.

USENIX Summer Conference
June 11-15, 1990
Anaheim, California

John K. Ousterhout – University of California at Berkeley and DEC Western Research Laboratory

Why Aren't Operating Systems Getting Faster As Fast as Hardware?

ABSTRACT

This paper evaluates several hardware platforms and operating systems using a set of benchmarks that stress kernel entry/exit, file systems, and other things related to operating systems. The overall conclusion is that operating system performance is not improving at the same rate as the base speed of the underlying hardware. The most obvious ways to remedy this situation are to improve memory bandwidth and reduce operating systems' tendency to wait for disk operations to complete.

1. Introduction

In the summer and fall of 1989 I assembled a collection of operating system benchmarks. My original intent was to compare the performance of Sprite, a UNIX-compatible research operating system developed at the University of California at Berkeley [4,5], with vendor-supported versions of UNIX running on similar hardware. After running the benchmarks on several configurations I noticed that the "fast" machines didn't seem to be running the benchmarks as quickly as I would have guessed from what I knew of the machines' processor speeds. In order to test whether this was a fluke or a general trend, I ran the benchmarks on a large number of hardware and software configurations at DEC's Western Research Laboratory, U.C. Berkeley, and Carnegie-Mellon University. This paper presents the results from the benchmarks.

Figure 1 summarizes the final results, which are consistent with my early observations: as raw CPU power increases, the speed of operating system functions (kernel calls, I/O operations, data moving) does not seem to be keeping pace. I found this to be true across a range of hardware platforms and operating systems. Only one "fast" machine, the VAX 8800, was able to execute a variety of benchmarks at speeds nearly commensurate with its CPU power, but the 8800 is not a particularly fast machine by today's standards and even it did not provide consistently good performance. Other machines ran from 10% to 10x more slowly (depending on the benchmark) than CPU power would suggest.

The benchmarks suggest at least two possible factors for the disappointing operating system performance: memory bandwidth and disk I/O. RISC architectures have allowed CPU speed to scale much faster than memory bandwidth, with the result that memory-intensive benchmarks do not receive the full benefit of faster CPUs. The second problem is in file systems. UNIX file systems require disk writes before several key operations can complete. As a result, the performance of those operations, and the performance of the file systems in general, are closely tied to disk speed and do not improve much with faster CPUs.

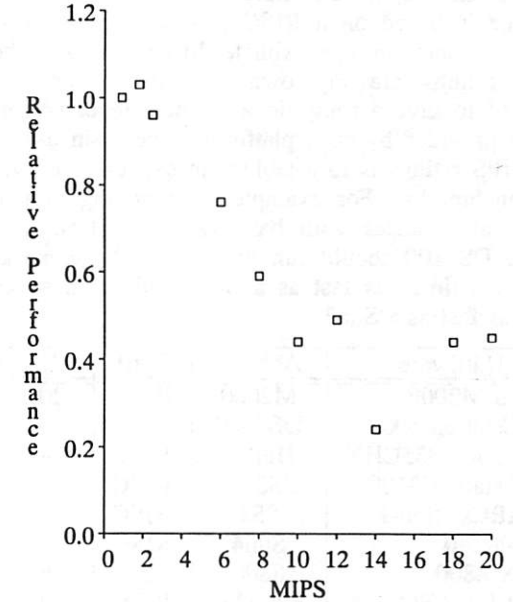


Figure 1. Summary of the results: operating system speed is not scaling at the same rate as basic hardware speed. Each point is the geometric mean of all the MIPS-relative performance numbers for all benchmarks on all operating systems on a particular machine. A value of 1.0 means that the operating system speed was commensurate with the machine's basic integer speed. The faster machines all have values much less than 1.0, which means that operating system functions ran more slowly than the machines' basic speeds would indicate.

The benchmarks that I ran are mostly "microbenchmarks": they measure specific features of the hardware or operating system (such as memory-to-memory copy speed or kernel entry-exit). Microbenchmarks make it easy to identify particular strengths and weaknesses of systems, but they do not usually provide a good overall indication of system performance. I also ran one "macro-benchmark" which exercises a variety of operating system features; this benchmark gives a better idea of overall system speed but does not provide much information about why some platforms are better than others. I make no guarantees that the collection of benchmarks discussed here is complete; it is possible that a different set of benchmarks might yield different results.

The rest of the paper is organized as follows. Section 2 gives a brief description of the hardware platforms and Section 3 introduces the operating systems that ran on those platforms. Sections 4-11 describe the eight benchmarks and the results of running them on various hardware-OS combinations. Section 12 concludes with some possible considerations for hardware and software designers.

2. Hardware

Table 1 lists the ten hardware configurations used for the benchmarks. It also includes an abbreviation for each configuration, which is used in the rest of the paper, an indication of whether the machine is based on a RISC processor or a CISC processor, and an approximate MIPS rating. The MIPS ratings are my own estimates; they are intended to give a rough idea of the integer performance provided by each platform. The main use of the MIPS ratings is to establish an expectation level for benchmarks. For example, if operating system performance scales with base system performance, then a DS3100 should run the various benchmarks about 1.5 times as fast as a Sun4 and about seven times as fast as a Sun3.

Hardware	Abbrev.	Type	MIPS
MIPS M2000	M2000	RISC	20
DECstation 5000	DS5000	RISC	18
H-P 9000-835CHX	HP835	RISC	14
DECstation 3100	DS3100	RISC	12
SPARCstation-1	SS1	RISC	10
Sun-4/280	Sun4	RISC	8
VAX 8800	8800	CISC	6
IBM RT-APC	RT-APC	RISC	2.5
Sun-3/75	Sun3	CISC	1.8
Microvax II	MVAX2	CISC	0.9

Table 1. Hardware platforms on which the benchmarks were run. "MIPS" is an estimate of integer CPU performance, where a VAX-11/780 is approximately 1.0.

All of the machines were generously endowed with memory. No significant paging occurred in any of the benchmarks. In the file-related benchmarks, the relevant files all fit in the main-memory buffer caches maintained by the operating systems. The machines varied somewhat in their disk configurations, so small differences in I/O-intensive benchmarks should not be considered significant. However, all measurements for a particular machine used the same disk systems, except that the Mach DS3100 measurements used different (slightly faster) disks than the Sprite and Ultrix DS3100 measurements.

The set of machines in Table 1 reflects the resources available to me at the time I ran the benchmarks. It is not intended to be a complete list of all interesting machines, but it does include a fairly broad range of manufacturers and architectures.

3. Operating Systems

I used six operating systems for the benchmarks: Ultrix, SunOS, RISC/os, HP-UX, Mach, and Sprite. Ultrix and SunOS are the DEC and Sun derivatives of Berkeley's 4.3 BSD UNIX, and are similar in many respects. RISC/os is MIPS Computer Systems' operating system for the M2000 machine. It appears to be a derivative of System V with some BSD features added. HP-UX is Hewlett-Packard's UNIX product and contains a combination of System V and BSD features. Mach is a new UNIX-like operating system developed at Carnegie Mellon University [1]. It is compatible with UNIX, and much of the kernel (the file system in particular) is derived from BSD UNIX. However, many parts of the kernel, including the virtual memory system and interprocess communication, have been re-written from scratch.

Sprite is an experimental operating system developed at U.C. Berkeley [4,5]; although it provides the same user interface as BSD UNIX, the kernel implementation is completely different. In particular, Sprite's file system is radically different from that of Ultrix and SunOS, both in the ways it handles the network and in the ways it handles disks. Some of the differences are visible in the benchmark results.

The version of SunOS used for Sun4 measurements was 4.0.3, whereas version 3.5 was used for Sun3 measurements. SunOS 4.0.3 incorporates a major restructuring of the virtual memory system and file system; for example, it maps files into the virtual address space rather than keeping them in a separate buffer cache. This difference is reflected in some of the benchmark results.

4. Kernel Entry-Exit

The first benchmark measures the cost of entering and leaving the operating system kernel. It does this by repeatedly invoking the *getpid* kernel call and taking the average time per invocation. *Getpid* does nothing but return the caller's process identifier. Table 2 shows the average time for this call on different platforms and operating systems.

The third column in the table is labeled "MIPS-Relative Speed". This column indicates how well the machine performed on the benchmark, relative to its MIPS rating in Table 1 and to the MVAX2 time in Table 2. Each entry in the third column was computed by taking the ratio of the MVAX2 time to the particular machine's time, and dividing that by the ratio of the machine's MIPS rating to the MVAX2's MIPS rating. For example, the DS5000 time for *getpid* was 11 microseconds, and its MIPS rating is approximately 18. Thus its MIPS-relative speed is (207/11)/(18/0.9) = 0.94. A MIPS-relative speed of 1.0 means that the given machine ran the benchmark at just the speed that would be expected based on the MVAX2 time and the MIPS ratings from Table 1. A MIPS-relative speed less than 1.0 means that the machine ran this benchmark more slowly than would be expected from its MIPS rating, and a figure larger than 1.0 means the machine performed better than might be expected.

Configuration	Time (μsec)	MIPS-Relative Speed
DS5000 Ultrix 3.1D	11	0.94
M2000 RISC/os 4.0	18	0.52
DS3100 Mach 2.5	23	0.68
DS3100 Ultrix 3.1	25	0.62
DS3100 Sprite	26	0.60
8800 Ultrix 3.0	28	1.1
SS1 SunOS 4.0.3	31	0.60
SS1 Sprite	32	0.58
Sun4 Mach 2.5	32	0.73
Sun4 SunOS 4.0.3	32	0.73
Sun4 Sprite	32	0.73
HP835 HP-UX	45	0.30
Sun3 Sprite	92	1.1
Sun3 SunOS 3.5	108	0.96
RT-APC Mach 2.5	148	0.50
MVAX2 Ultrix 3.0	207	1.0

Table 2. Time for the *getpid* kernel call. "MIPS-Relative Speed" normalizes the benchmark speed to the MIPS ratings in Table 1: a MIPS-relative speed of .5 means the machine ran the benchmark only half as fast as would be expected by comparing the machine's MIPS rating to the MVAX2. The HP835 measurement is for *gethostid* instead of *getpid*: *getpid* appears to cache the process id in user space, thereby avoiding kernel calls on repeated invocations (the time for *getpid* was 4.3 microseconds).

Although the RISC machines were generally faster than the CISC machines on an absolute scale, they were not as fast as their MIPS ratings would suggest: their MIPS-relative speeds were typically in the range 0.5 to 0.8. This indicates that the cost of entering and exiting the kernel in the RISC machines has not improved as much as their basic computation speed.

5. Context Switching

The second benchmark is called *cswitch*. It measures the cost of context switching plus the time for processing small pipe reads and writes. The benchmark operates by forking a child process and then repeatedly passing one byte back and forth between parent and child using two pipes (one for each direction). Table 3 lists the average time for each round-trip between the processes, which includes one *read* and one *write* kernel call in each process, plus two context switches. As with the *getpid* benchmark, MIPS-relative speeds were computed by scaling from the MVAX2 times and the MIPS ratings in Table 1.

Once again, the RISC machines were generally faster than the CISC machines, but their MIPS-relative speeds were only in the range of 0.3 to 0.5. The only exceptions occurred with Ultrix on the DS3100, which had a MIPS-relative speed of about 0.81, and with Ultrix on the DS5000, which had a MIPS-relative speed of 1.02.

Configuration	Time (ms)	MIPS-Relative Speed
DS5000 Ultrix 3.1D	0.18	1.02
M2000 RISC/os 4.0	0.30	0.55
DS3100 Ultrix 3.1	0.34	0.81
DS3100 Mach 2.5	0.50	0.55
DS3100 Sprite	0.51	0.54
8800 Ultrix 3.0	0.70	0.78
Sun4 Mach 2.5	0.82	0.50
Sun4 SunOS 4.0.3	1.02	0.40
SS1 SunOS 4.0.3	1.06	0.32
HP835 HP-UX	1.12	0.21
Sun4 Sprite	1.17	0.35
SS1 Sprite	1.19	0.28
Sun3 SunOS 3.5	2.36	0.78
Sun3 Sprite	2.41	0.76
RT-APC Mach 2.5	3.52	0.37
MVAX2 Ultrix 3.0	3.66	1.0

Table 3. Context switching costs, measured as the time to echo one byte back and forth between two processes using pipes.

6. Select

The third benchmark exercises the *select* kernel call. It creates a number of pipes, places data in some of those pipes, and then repeatedly calls *select* to determine how many of the pipes are readable. A zero timeout is used in each *select* call so that the kernel call never waits. Table 4 shows how long each *select* call took, in microseconds, for three configurations. Most of the commercial operating systems (SunOS, Ultrix, and RISC/os) gave performance generally in line with the machines' MIPS ratings, but HP-UX and the two research operating systems (Sprite and Mach) were somewhat slower than the others.

The M2000 numbers in Table 4 are surprisingly high for pipes that were empty, but quite low as long as at least one of the pipes contained data. I suspect that RISC/os's emulation of the *select* kernel call is faulty and causes the process to wait for 10 ms even if the calling program requests immediate timeout.

7. Block Copy

The fourth benchmark uses the *bcopy* procedure to transfer large blocks of data from one area of memory to another. It doesn't exercise the operating system at all, but different operating systems differ for the same hardware because their libraries contain different *bcopy* procedures. The main differences, however, are due to the cache organizations and memory bandwidths of the different machines.

The results are given in Table 5. For each configuration I ran the benchmark with two different block sizes. In the first case, I used blocks large enough (and aligned properly) to use *bcopy* in the most efficient way possible. At the same time the blocks were small enough that both the source and destination block fit in the cache (if any). In the second case I used a transfer size larger than the cache size, so that cache misses occurred. In each case several transfers were made between the same source and destination, and the average bandwidth of copying is shown in Table 5.

The last column in Table 5 is a relative figure showing how well each configuration can move large uncached blocks of memory relative to how fast it executes normal instructions. I computed this figure by taking the number from the second column ("Uncached") and dividing it by the MIPS rating

Configuration	1 pipe (*μsec*)	10 empty (*μsec*)	10 full (*μsec*)	MIPS-Relative Speed
DS5000 Ultrix 3.1D	44	91	90	1.01
M2000 RISC/os 4.0	10000	10000	108	0.76
DS3100 Sprite	76	240	226	0.60
DS3100 Ultrix 3.1	81	153	151	0.90
DS3100 Mach 2.5	95	178	166	0.82
Sun4 SunOS 4.0.3	103	232	213	0.96
SS1 SunOS 4.0.3	110	221	204	0.80
8800 Ultrix 3.0	120	265	310	0.88
HP835 HP-UX	122	227	213	0.55
Sun4 Sprite	126	396	356	0.58
SS1 Sprite	138	372	344	0.48
Sun4 Mach 2.5	150	300	266	0.77
Sun3 Sprite	413	1840	1700	0.54
Sun3 SunOS 3.5	448	1190	1012	0.90
RT-APC Mach 2.5	701	1270	1270	0.52
MVAX2 Ultrix 3.0	740	1610	1820	1.0

Table 4. Time to execute a *select* kernel call to check readability of one or more pipes. In the first column a single empty pipe was checked. In the second column ten empty pipes were checked, and in the third column the ten pipes each contained data. The last column contains MIPS-relative speeds computed from the "10 full" case.

Configuration	Cached (Mbytes/second)	Uncached (Mbytes/second)	Mbytes/MIPS
DS5000 Ultrix 3.1D	40	12.6	0.70
M2000 RISC/os 4.0	39	20	1.00
8800 Ultrix 3.0	22	16	2.7
HP835 HP-UX	17.4	6.2	0.44
Sun4 Sprite	11.1	5.0	0.55
SS1 Sprite	10.4	6.9	0.69
DS3100 Sprite	10.2	5.4	0.43
DS3100 Mach 2.5	10.2	5.1	0.39
DS3100 Ultrix 3.1	10.2	5.1	0.39
Sun4 SunOS 4.0.3	8.2	4.7	0.52
Sun4 Mach 2.5	8.1	4.6	0.56
SS1 SunOS 4.0.3	7.6	5.6	0.56
RT-APC Mach 2.5	5.9	5.9	2.4
Sun3 Sprite	5.6	5.5	3.1
MVAX2 Ultrix 3.0	3.5	3.3	3.7

Table 5. Throughput of the *bcopy* procedure when copying large blocks of data. In the first column the data all fit in the processor's cache (if there was one) and the times represent a "warm start" condition. In the second column the block being transferred was much larger than the cache.

from Table 1. Thus, for the 8800 the value is (16/6) = 2.7.

The most interesting thing to notice is that the CISC machines (8800, Sun3, and MVAX2) have normalized ratings of 2.7-3.7, whereas all of the RISC machines except the RT-APC have ratings of 1.0 or less. Memory-intensive applications are not likely to scale well on these RISC machines. In fact, the relative performance of memory copying drops almost monotonically with faster processors, both for RISC and CISC machines.

The relatively poor memory performance of the RISC machines is just another way of saying that the RISC approach permits much faster CPUs for a given memory system. An inevitable result of this is that memory-intensive applications will not benefit as much from RISC architectures as non-memory-intensive applications.

Configuration	MB/sec.	MIPS-Relative Speed
M2000 RISC/os 4.0	15.6	0.31
8800 Ultrix 3.0	10.5	0.68
Sun4 SunOS 4.0.3	8.9	0.44
Sun4 Mach 2.5	6.8	0.33
Sun4 Sprite	6.8	0.33
SS1 SunOS 4.0.3	6.3	0.25
DS5000 Ultrix 3.1D	6.1	0.13
SS1 Sprite	5.9	0.23
HP835 HP-UX	5.8	0.16
DS3100 Mach 2.5	4.8	0.16
DS3100 Ultrix 3.1	4.8	0.16
DS3100 Sprite	4.4	0.14
RT-APC Mach 2.5	3.7	0.58
Sun3 Sprite	3.7	0.80
Sun3 SunOS 3.5	3.1	0.67
MVAX2 Ultrix 3.0	2.3	1.0

Table 6. Throughput of the *read* kernel call when reading large blocks of data from a file small enough to fit in the main-memory file cache.

8. Read from File Cache

The *read* benchmark opens a large file and reads the file repeatedly in 16-kbyte blocks. For each configuration I chose a file size that would fit in the main-memory file cache. Thus the benchmark measures the cost of entering the kernel and copying data from the kernel's file cache back to a buffer in the benchmark's address space. The file was large enough that the data to be copied in each kernel call was not resident in any hardware cache. However, the same buffer was re-used to receive the data from each call; in machines with caches, the receiving buffer was likely to stay in the cache. Table 6 lists the overall bandwidth of data transfer, averaged across a large number of kernel calls.

The numbers in Table 6 reflect fairly closely the memory bandwidths from Table 5. The only noticeable differences are for the Sun4 and the DS5000. The Sun4 does relatively better in this benchmark due to its write-back cache. Since the receiving buffer always stays in the cache, its contents get overwritten without ever being flushed to memory. The other machines all had write-through caches, which caused information in the buffer to be flushed immediately to memory. The second difference from Table 5 is for the DS5000, which was much slower than expected; I do not have an explanation for this discrepancy.

9. Modified Andrew Benchmark

The only large-scale benchmark in my test suite is a modified version of the Andrew benchmark developed by M. Satyanarayanan for measuring the performance of the Andrew file system [3]. The benchmark operates by copying a directory hierarchy containing the source code for a program, *stat*-ing every file in the new hierarchy, reading the contents of every copied file, and finally compiling the code in the copied hierarchy.

Satyanarayanan's original benchmark used whichever C compiler was present on the host machine, which made it impossible to compare running times between machines with different architectures or different compilers. In order to make the results comparable between different machines, I modified the benchmark so that it always uses the same compiler, which is the GNU C compiler generating code for an experimental machine called SPUR [2].

Table 7 contains the raw Andrew results. The table lists separate times for two different phases of the benchmark. The "copy" phase consists of everything except the compilation (all of the file copying and scanning), and the "compile" phase consists of just the compilation. The copy phase is much more I/O-intensive than the compile phase, and it also makes heavy use of the mechanisms for process creation (for example, a separate shell command is used to copy each file). The compile phase is CPU-bound on the slower machines, but spends a significant fraction of time in I/O on the faster machines.

I ran the benchmark in both local and remote configurations. "Local" means that all the files accessed by the benchmark were stored on a disk attached to the machine running the benchmark. "Diskless" refers to Sprite configurations where the machine running the benchmark had no local disk; all files, including the program binaries, the files in the directory tree being copied and compiled, and temporary files were accessed over the network using the Sprite file system protocol [4]. "NFS" means that the NFS protocol [7] was used to access remote files. For the SunOS NFS measurements the machine running the benchmark was diskless. For the Ultrix and Mach measurements the machine running the benchmark had a local disk that was used for temporary files, but all other files were accessed remotely. "AFS" means that the Andrew file system protocol was used for accessing remote files [3]; this configuration was similar to NFS under Ultrix in that the machine running the benchmark had a local disk, and temporary files were stored on that local disk while other files were accessed remotely. In each case the server was the same kind of machine as the client.

Table 8 gives additional "relative" numbers: the MIPS-relative speed for the local case, the MIPS-relative speed for the remote case, and the percentage slow-down experienced when the benchmark ran with a remote disk instead of a local one.

There are several interesting results in Tables 7 and 8. First of all, the faster machines generally have smaller MIPS-relative speeds than the slower

Configuration	Copy (seconds)	Compile (seconds)	Total (seconds)
M2000 RISC/os 4.0 Local	13	59	72
DS3100 Sprite Local	22	98	120
DS5000 Ultrix 3.1D Local	48	76	124
DS3100 Sprite Diskless	34	93	127
DS3100 Mach 2.5 Local	29	107	136
Sun4 Mach 2.5 Local	37	122	159
Sun4 Sprite Local	44	128	172
Sun4 Sprite Diskless	56	128	184
DS5000 Ultrix 3.1D NFS	68	118	186
Sun4 SunOS 4.0.3 Local	54	133	187
SS1 SunOS 4.0.3 Local	54	139	193
DS3100 Mach 2.5 NFS	58	147	205
DS3100 Ultrix 3.1 Local	80	133	213
8800 Ultrix 3.0 Local	48	181	229
SS1 SunOS 4.0.3 NFS	76	168	244
DS3100 Ultrix 3.1 NFS	115	154	269
Sun4 SunOS 4.0.3 NFS	92	213	305
Sun3 Sprite Local	52	375	427
RT-APC Mach 2.5 Local	89	344	433
Sun3 Sprite Diskless	75	364	439
Sun3 SunOS 3.5 Local	69	406	475
RT-APC Mach 2.5 AFS	128	397	525
Sun3 SunOS 3.5 NFS	157	478	635
MVAX2 Ultrix 3.0 Local	214	1202	1416
MVAX2 Ultrix 3.0 NFS	298	1409	1707

Table 7. Elapsed time to execute a modified version of M. Satyanarayanan's Andrew benchmark [3]. The first column gives the total time for all of the benchmark phases except the compilation phase. The second column gives the elapsed time for compilation, and the third column gives the total time for the benchmark. The entries in the table are ordered by total execution time.

Configuration	MIPS-Relative Speed (Local)	MIPS-Relative Speed (Remote)	Remote Penalty (%)
M2000 RISC/os 4.O Local	0.88	--	--
8800 Ultrix	0.93	--	--
Sun-4 Mach 2.5	1.0	--	--
Sun3 Sprite	1.7	1.9	3
DS3100 Sprite	0.89	1.01	6
Sun4 Sprite	0.93	1.04	7
MVAX2 Ultrix 3.0 NFS	1.0	1.0	21
RT-APC Mach 2.5 AFS	1.2	1.2	21
DS3100 Ultrix 3.1 NFS	0.50	0.48	26
SS1 SunOS 4.0.3 NFS	0.66	0.63	26
Sun3 SunOS 3.5 NFS	1.4	1.3	34
DS3100 Mach 2.5 NFS	0.78	0.62	50
DS5000 Ultrix 3.1D NFS	0.57	0.46	50
Sun4 SunOS 4.0.3 NFS	0.85	0.63	63

Table 8. Relative performance of the Andrew benchmark. The first two columns give the MIPS-relative speed, both local and remote. The first column is computed relative to the MVAX2 Ultrix 3.0 Local time and the second column is computed relative to MVAX2 Ultrix 3.0 NFS. The third column gives the remote penalty, which is the additional time required to execute the benchmark remotely, as a percentage of the time to execute it locally. The entries in the table are ordered by remote penalty.

machines. This is easiest to see by comparing different machines running the same operating system, such as Sprite on the Sun3, Sun4 and DS3100 (1.7, 0.93, and 0.89 respectively for the local case) or SunOS on the Sun3, Sun4, and SS1 (1.4, 0.85, 0.66 respectively for the local case) or Ultrix on the MVAX2, 8800, and DS3100 (1.0, 0.93, and 0.50 respectively for the local case).

The second overall result from Tables 7 and 8 is that Sprite is faster than other operating systems in every case except in comparison to Mach on the Sun4. For the local case Sprite was generally 10-20% faster, but in the remote case Sprite was typically 30-70% faster. On Sun-4's and DS3100's, Sprite was 60-70% faster than either Ultrix, SunOS, or Mach for the remote case. In fact, the Sprite-DS3100 combination ran the benchmark remotely 45% faster than Ultrix-DS5000, even though the Ultrix-DS5000 combination had about 50% more CPU power to work with. Table 8 shows that Sprite ran the benchmark almost as fast remotely as locally, whereas the other systems slowed down by 20-60% when running remotely with NFS or AFS. It appears that the penalty for using NFS is increasing as machine speeds increase: the MVAX2 had a remote penalty of 21%, the Sun3 34%, and the Sun4 63%.

The third interesting result of this benchmark is that the DS3100-Ultrix-Local combination is slower than I had expected: it is about 24% slower than DS3100-Mach-Local and 78% slower than DS3100-Sprite-Local. The DS3100-Ultrix combination did not experience as great a remote penalty as other configurations, but this is because the local time is unusually slow.

10. Open-Close

The modified Andrew benchmark suggests that the Sprite file system is faster than the other file systems, particularly for remote access, but it doesn't identify which file system features are responsible. I ran two other benchmarks in an attempt to pinpoint the differences. The first benchmark is open-close, which repeatedly opens and closes a single file. Table 9 displays the cost of an open-close pair for two cases: a name with only a single element, and one with 4 elements. In both the local and remote cases the UNIX derivatives are consistently faster than Sprite. The remote times are dominated by the costs of server communication: Sprite communicates with the server on every open or close, NFS occasionally communicates with the server (to check the consistency of its cached naming information), and AFS virtually never checks with the server (the

Configuration	''foo'' (ms)	''a/b/c/foo'' (ms)	MIPS-Relative Speed
DS5000 Ultrix 3.1D Local	0.16	0.31	0.91
DS3100 Mach 2.5 Local	0.19	0.33	1.1
Sun4 SunOS 4.0.3 Local	0.25	0.38	1.3
DS3100 Ultrix 3.1 Local	0.27	0.41	0.81
Sun4 Mach 2.5 Local	0.30	0.40	1.1
SS1 SunOS 4.0.3 Local	0.31	0.44	0.84
M2000 RISC/os 4.0 Local	0.32	0.83	0.41
HP835 HP-UX Local	0.38	0.61	0.49
8800 Ultrix 3.0 Local	0.45	0.68	0.97
DS3100 Sprite Local	0.82	0.97	0.27
RT-APC Mach 2.5 Local	0.95	1.6	1.1
Sun3 SunOS 3.5 Local	1.1	2.2	1.3
Sun4 Sprite Local	1.2	1.4	0.27
RT-APC Mach 2.5 AFS	1.7	3.5	7.6
DS5000 Ultrix 3.1D NFS	2.4	2.4	0.75
MVAX2 Ultrix 3.0 Local	2.9	4.7	1.0
SS1 SunOS 4.0.3 NFS	3.4	3.5	0.95
Sun4 SunOS 4.0.3 NFS	3.5	3.7	1.2
DS3100 Mach 2.5 NFS	3.6	3.9	0.75
DS3100 Ultrix 3.1 NFS	3.8	3.9	0.71
DS3100 Sprite Diskless	4.3	4.4	0.63
Sun3 Sprite Local	4.3	5.2	0.34
Sun4 Sprite Diskless	6.1	6.4	0.67
HP835 HP-UX NFS	7.1	7.3	0.33
Sun3 SunOS 3.5 NFS	10.4	11.4	1.7
Sun3 Sprite Diskless	12.8	16.3	1.4
MVAX2 Ultrix 3.0 NFS	36.0	36.9	1.0

Table 9. Elapsed time to open a file and then close it again, using the *open* and *close* kernel calls. The MIPS-relative speeds are for the ''foo'' case, scaled relative to MVAX2 Ultrix 3.0 Local for local configurations and relative to MVAX2 Ultrix 3.0 NFS for remote configurations.

server must notify the client if any cached information becomes invalid). Because of its ability to avoid all server interactions on repeated access to the same file, AFS was by far the fastest remote file system for this benchmark.

Although this benchmark shows dramatic differences in open-close costs, it does not seem to explain the performance differences in Table 8. The MIPS-relative speeds vary more from operating system to operating system than from machine to machine. For example, all the Mach and Ultrix MIPS-relative speeds were in the range 0.8 to 1.1 for the local case, whereas all the Sprite MIPS-relative speeds were in the range 0.27 to 0.34 for the local case.

11. Create-Delete

The last benchmark was perhaps the most interesting in terms of identifying differences between operating systems. It also helps to explain the results in Tables 7 and 8. This benchmark simulates the creation, use, and deletion of a temporary file. It opens a file, writes some amount of data to the file, and closes the file. Then it opens the file for reading, reads the data, closes the file, and finally deletes the file. I tried three different amounts of data: none, 10 kbytes, and 100 kbytes. Table 10 gives the total time to create, use, and delete the file in each of several hardware/operating system configurations.

This benchmark highlights a basic difference between Sprite and the UNIX derivatives. In Sprite, short-lived files can be created, used, and deleted without any data ever being written to disk. Information only goes to disk after it has lived at least 30 seconds. Sprite requires only a single disk I/O for each iteration of the benchmark, to write out the file's i-node after it has been deleted. Thus in the best case (DS3100's) each iteration takes one disk rotation, or about 16 ms. Even this one I/O is an historical artifact that is no longer necessary; a newer version of the Sprite file system eliminates it, resulting in a benchmark time of only 4 ms for the DS3100-Local-No-Data case.

UNIX and its derivatives are all much more disk-bound than Sprite. When files are created and deleted, several pieces of information must be forced to disk and the operations cannot be completed until the I/O is complete. Even with no data in the file, the UNIX derivatives all required 35-100 ms to

Configuration	No data (ms)	10 kbytes (ms)	100 kbytes (ms)	MIPS-Relative Speed
DS3100 Sprite Local	17	34	69	0.44
Sun4 Sprite Local	18	33	67	0.63
DS3100 Sprite Remote	33	34	68	0.67
Sun3 Sprite Local	33	47	130	1.5
M2000 RISC/os 4.0 Local	33	51	116	0.12
Sun4 Sprite Remote	34	50	71	0.98
8800 Ultrix 3.0 Local	49	100	294	0.31
DS5000 Ultrix 3.1D Local	50	86	389	0.09
Sun4 Mach 2.5 Local	50	83	317	0.23
DS3100 Mach 2.5 Local	50	100	317	0.15
HP835 HP-UX Local	50	115	263	0.13
RT-APC Mach 2.5 Local	53	121	706	0.68
Sun3 Sprite Remote	61	73	129	2.42
SS1 SunOS 4.0.3 Local	65	824	503	0.14
Sun4 SunOS 4.0.3 Local	67	842	872	0.17
Sun3 SunOS 3.5 Local	67	105	413	0.75
DS3100 Ultrix 3.1 Local	80	146	548	0.09
SS1 SunOS 4.0.3 NFS	82	214	1102	0.32
DS5000 Ultrix 3.1D NFS	83	216	992	0.18
DS3100 Mach 2.5 NFS	89	233	1080	0.25
Sun4 SunOS 4.0.3 NFS	97	336	2260	0.34
MVAX2 Ultrix 3.0 Local	100	197	841	1.0
DS3100 Ultrix 3.1 NFS	116	370	3028	0.19
RT-APC Mach 2.5 AFS	120	303	1615	0.89
Sun3 SunOS 3.5 NFS	152	300	1270	0.97
HP835 HP-UX NFS	180	376	1050	0.11
MVAX2 Ultrix 3.0 NFS	295	634	2500	1.0

Table 10. Elapsed time to create a file, write some number of bytes to it, close the file, then re-open the file, read it, close it, and delete it. This benchmark simulates the use of a temporary file. The Mach-AFS combination showed great variability: times as high as 460ms/721ms/2400ms were as common as the times reported above (the times in the table were the lowest ones seen). MIPS-relative speeds were computed using the "No Data" times in comparison to the MVAX2 local or NFS time. The table is sorted in order of "No Data" times.

create and delete the file. This suggests that information like the file's i-node, the entry in the containing directory, or the directory's i-node is being forced to disk. In the NFS-based remote systems, newly-written data must be transferred over the network to the file server and then to disk before the file may be closed. Furthermore, NFS forces each block to disk independently, writing the file's i-node and any dirty indirect blocks once for each block in the file. This results in up to 3 disk I/O's for each block in the file. In AFS, modified data is returned to the file server as part of the close operation. The result of all these effects is that the performance of the create-delete benchmark under UNIX (and the performance of temporary files under UNIX) are determined more by the speed of the disk than by the speed of the CPU.

The create-delete benchmark helps to explain the poor performance of DS3100 Ultrix on the Andrew benchmark. The basic time for creating an empty file is 60% greater in DS3100-Ultrix-Local than in 8800-Ultrix-Local, even though the DS3100 CPU is twice as fast as the 8800 CPU. The time for a 100-kbyte file in DS3100-Ultrix-NFS is 45 times as long as for DS3100-Sprite-Diskless! The poor performance relative to the 8800 may be due in part to slower disks (RZ55's on the DS3100's). However, Ultrix's poor remote performance is only partially due to NFS's flush-on-close policy: DS3100-Ultrix-NFS achieves a write bandwidth of only about 30 kbytes/sec on 100 Kbyte files, which is almost twice as slow as I measured on the same hardware running an earlier version of Ultrix (3.0) and three times slower than Mach. I suspect that Ultrix could be tuned to provide substantially better file system performance.

Lastly, Table 10 exposes some surprising behavior in SunOS 4.0.3. The benchmark time for a file with no data is 67 ms, but the time for 10 kbytes is 842 ms, which is almost an order of magnitude slower than SunOS 3.5 running on a Sun3! This was so surprising that I also tried data sizes of 2-9 kbytes at 1-kbyte intervals. The SunOS 4.0.3 time stayed in the 60-80ms range until the file size increased from 8 kbytes to 9 kbytes; at this point it jumped up to about 800 ms. This anomaly is not present in other UNIX derivatives, or even in earlier versions of SunOS. Since the jump occurs at a file size equal to the page size, I hypothesize that it is related to the implementation of mapped files in SunOS 4.0.3.

12. Conclusions

For almost every benchmark the faster machines ran more slowly than I would have guessed from raw processor speed. Although it is dangerous to draw far-reaching conclusions from a small set of benchmarks, I think that the benchmarks point out four potential problem areas, two for hardware designers and two for operating system developers.

12.1 Hardware Issues

The first hardware-related issue is memory bandwidth: the benchmarks suggest that it is not keeping up with CPU speed. Part of this is due to the 3-4x difference in CPU speed relative to memory bandwidth in newer RISC architectures versus older CISC architectures; this is a one-time-only effect that occurred in the shift from RISC to CISC. However, I believe it will be harder for system designers to improve memory performance as fast as they improve processor performance in the years to come. In particular, workstations impose severe cost constraints that may encourage designers to skimp on memory system performance. If memory bandwidth does not improve dramatically in future machines, some classes of applications may be limited by memory performance.

A second hardware-related issue is context switching. The *getpid* and *cswitch* benchmarks suggest that context switching, both for kernel calls and for process switches, is about 2x more expensive in RISC machines than in CISC machines. I don't have a good explanation for this result, since the extra registers in the RISC machines cannot account for the difference all by themselves. A 2x degradation may not be serious, as long as the relative performance of context switching doesn't degrade any more in future machines.

12.2 Software Issues

In my view, one of the greatest challenges for operating system developers is to decouple file system performance from disk performance. Operating systems derived from UNIX use caches to speed up reads, but they require synchronous disk I/O for operations that modify files. If this coupling isn't eliminated, a large class of file-intensive programs will receive little or no benefit from faster hardware. Of course, delaying disk writes may result in information loss during crashes; the challenge for operating system designers is to maintain an acceptable level of reliability while decoupling performance.

One approach that is gaining in popularity is to use non-volatile memory as a buffer between main memory and disk. Information can be written immediately (and efficiently) to the non-volatile memory so that it will survive crashes; long-lived information can eventually be written to disk. This approach involves extra complexity and overhead to move information first to non-volatile memory and then to disk, but it may result in better overall performance than writing immediately to disk. Another new approach is to use log-structured file systems, which decouple file system performance from disk performance and make disk I/O's more efficient. See [6] for details.

A final consideration is in the area of network protocols. In my opinion, the assumptions inherent in NFS (statelessness and write-through-on-close, in particular) represent a fundamental performance limitation. If users are to benefit from faster machines, either NFS must be scrapped (my first choice), or NFS must be changed to be less disk-intensive.

13. Code Availability

The source code for all of these benchmarks is available via public FTP from ucbvax.berkeley.edu. The file `pub/mab.tar.Z` contains the modified Andrew benchmark and `pub/bench.tar.Z` contains all of the other benchmarks.

14. Acknowledgments

M. Satyanarayanan developed the original Andrew benchmark and provided me with access to an IBM RT-APC running Mach. Jay Kistler resolved the incompatibilities that initially prevented the modified Andrew benchmark from running under Mach. Jeff Mogul and Paul Vixie helped me get access to DEC machines and kernels for testing, and explained the intricacies of configuring NFS. Rick Rashid provided me with benchmark results for Mach running on DS3100's and Sun4's. Joel McCormack and David Wall provided helpful comments on earlier drafts of this paper.

15. References

[1] Accetta, M., et al. "Mach: A New Kernel Foundation for UNIX Development." *Proceedings of the USENIX 1986 Summer Conference*, July 1986, pp. 93-113.

[2] Hill, M., et al. "Design Decisions in SPUR." *IEEE Computer*, Vol. 19, No. 11, November 1986, pp. 8-22.

[3] Howard, J., et al. "Scale and Performance in a Distributed File System." *ACM Transactions on Computer Systems*, Vol. 6, No. 1, February 1988, pp. 51-81.

[4] Nelson, M., Welch, B., and Ousterhout, J. "Caching in the Sprite Network File System." *ACM Transactions on Computer Systems*, Vol. 6, No. 1, February 1988 pp. 134-154.

[5] Ousterhout, J., et al. "The Sprite Network Operating System." *IEEE Computer*, Vol. 21, No. 2, February 1988, pp. 23-36.

[6] Rosenblum, M., and Ousterhout, J. "The LFS Storage Manager." *Proceedings of the USENIX 1990 Summer Conference*, June 1990.

[7] Sandberg, R., et al. "Design and Implementation of the Sun Network Filesystem." *Proceedings of the USENIX 1985 Summer Conference*, June 1985, pp. 119-130.

John K. Ousterhout is a Professor in the Department of Electrical Engineering and Computer Sciences at the University of California, Berkeley. His interests include operating systems, distributed systems, user interfaces, and computer-aided design. He is currently leading the development of Sprite, a network operating system for high-performance workstations. In the past, he and his students developed several widely-used programs for computer-aided design, including Magic, Caesar, and Crystal. Ousterhout is a recipient of the ACM Grace Murray Hopper Award, the National Science Foundation Presidential Young Investigator Award, the National Academy of Sciences Award for Initiatives in Research, the IEEE Browder J. Thompson Award, and the UCB Distinguished Teaching Award. He received a B.S. degree in Physics from Yale University in 1975 and a Ph.D. in Computer Science from Carnegie Mellon University in 1980. His electronic address is ouster@sprite.berkeley.edu.

USENIX Summer Conference
June 11-15, 1990
Anaheim, California

R. D. Trammel – Tektronix (Visual Systems Group)

The Big Picture: Visualizing System Behavior in Real Time

ABSTRACT

In developing UNIX(TM) systems and applications it is frequently useful to observe sys tem behavior and resource consumption. Several tools exist which provide information of this kind. KATO was developed to address the need for a performance analysis tool which had broader coverage, did not perturb the target system, and displayed data graphically in real time.

KATO solicits telemetry from a remote target and runs as an X client, presenting information through five basic display types. A flight recorder allows the storage and playback of telemetry via a VCR-like control panel. Some novel aspects of this tool are the variety and immediacy of data, techniques of visual presentation, selection of per process information, and the ability to remotely modify system tunable parameters.

1. Introduction

In 1989 a series of workstation virtual memory performance problems prompted the development of KATO (Kernel Analysis TOol), a visual aid for dealing with UNIX performance issues of all kinds. KATO was brought to bear on several VM problems and proved its utility immediately. It currently runs under SVR3.2 on XD88(TM) platforms and this discussion will be from a System V(TM) perspective.

Sophisticated performance tools will be required to develop the operating systems of the 90's. This paper deals with what we have learned regarding the design, implementation, and application of such a tool. In particular, it focuses on techniques of visual presentation and user control. The value of high sampling rates and real time issues associated with achieving those rates are discussed. At the conclusion, we shall attempt to characterize a "next generation" OS performance analyzer.

1.1 Domains of Performance Analysis

Often, in the heat of aggressive development schedules, little time is devoted to system performance issues beyond running the industry standard benchmarks. The customer may be the first to notice a performance problem. One opportunity to apply performance tools is during system development, to discover performance leaks, and to tune OS policies for the best performance under a set of representative loads.

Another domain is the optimization of applications. Authors of application code are often unaware of the impact of their demands for resources on the OS. This is particularly true of virtual memory and file buffers. Many applications can benefit from such scrutiny because their profligate use of a resource is unintentional.

The last domain we shall mention is the characterization of pathological behaviors. A good performance tool can occasionally contribute to debugging efforts by providing additional resource usage and state information in the beginning, though it will likely be useless in discovering a solution.

1.2 Tools of the Past: What's Good, What's Missing

An excellent performance monitoring tool, from the standpoint of completeness, is the sar family of utilities. Many of the statistics gathered for sar are also utilized by KATO. On the negative side, sar reports average and total figures over long time periods and presents its data in text form. Although sar has a broader scope than the other tools discussed in this section, its poor time resolution precludes information on high frequency events (e.g. process state transitions).

Profiler is a set of utilities which report on comparative time spent in the OS by function name. It is useful when minimizing system time is the issue, but gives no direct information on usage of other resources. Data are gathered by recording the PC when kernel execution is interrupted by the clock. Thus cycles consumed from the clock handler are not visible. Profiler requires intimate knowledge of OS routines and what they do in order to interpret its output. An analogous tool for measurements on user processes, prof, is available in both System V and BSD environments.

Xperfmon displays graphs of a small set of performance statistics to a specified time resolution in the seconds range. Its graphic format and real time presentation provided the original inspiration for

KATO. Unfortunately, xperfmon only provides data on CPU usage, free memory, disk bandwidth, interrupt frequency, and LAN traffic. Xperfmon is a public domain X Windows client. A similar graphic tool is available on SUN platforms.

The BSD utility systat provides a variety of statistics on virtual memory behavior, swap space, IO and interrupts per device, CPU per process, and network connections. Displays can be interactively selected or hidden by the user. Systat consumes 10% of a VAX to update the display at five-second intervals. Systat uses curses to update the numbers and character bargraphs.

Kernel debuggers (e.g., dbx, gdb) are occasionally applied to performance problems for lack of anything better. The snapshot nature of such measurements, high keystroke overhead, kernel expertise required, and textual format limit the utility of these tools. Their advantage is the ability to make some ad hoc measurements not covered by other tools.

AT&T distributes tunex, an unbundled software package which observes system behavior and makes recommendations concerning configuration and tuning. As an attempt to embed arcane expertise in a piece of software, it deserves some credit for innovation. Tunex is prejudiced toward System V disk and filesystem optimization. It offers no advice on tuning paging policy parameters, an area in which even system developers could use a hand.

2. Design Goals

KATO was implemented after two weeks of design work in which the following high level issues were considered.

2.1 Avoiding the Observer Effect

The act of making a measurement may render the measurement invalid because energy expended to probe the system perturbs its state. Of the tools discussed in section 1.2. earlier, those which offer any degree of real time measurement are plagued by high system overhead. A major design goal for KATO was to provide a finer time resolution than the other tools offered. It then became important to minimize the impact of KATO on the target machine. For this and other reasons discussed in section 2.5 it was decided that KATO would be divided into two parts: raw data collection running in target kernel code, and an X client process running on a separate machine to perform the data interpretation and display. The target would package telemetry and transmit it to the monitoring host via LAN.

2.2 Design Tradeoffs in Data Presentation

If enough information can be clearly presented, the viewer no longer sees a disjoint collection of facts and begins to perceive a more integrated picture of overall system phenomena. Reaching this "threshold of visualization" was the primary design goal for KATO.

Presentation methods were designed to avoid confusion and enhance the visibility of relationships. The decision of how to display an item was influenced by conflicting considerations:

- Screen space. A generalized performance analyzer must give access to more information than will fit on the physical screen in graphic form. Each display was chosen and sized to minimize area while preserving useful information.
- Display update rate. How often an item's display is updated depends on the rendering cost, how often the item changes, and how much time resolution is meaningful to the viewer. Process state change events should update the display whenever they happen. Percent time executing user code can be averaged over one second periods.
- Rendering time. Unfortunately, the most informative displays tend to be the most expensive. Trading off update rate was done whenever possible.
- Behaviors of interest. A display method should be chosen that best highlights the behaviors of interest at least expense; e.g., don't expend a chart recorder on an item which only takes on two values, use a color indicator instead. When patterns of behavior over time are important, or if correlations between statistics may be significant, then chart recorder displays should be used.

2.3 Selection of Data for Display

Given that all the available information will not fit on the screen and may exceed the rendering bandwidth as well, a means of selecting items for display is needed.

First, the information would be divided into subsets targeted to specific audiences: system developers and application engineers. Command line options control which format is selected.

Second, displays would be segregated into windows by topic. Since the windows are created only by user request, no graphics resources are consumed by uninteresting windows, and no cycles would be expended to interpret their data.

Finally, the user would be allowed to request a window containing more detailed information regard ing a process or resource.

2.4 Use of Color

KATO presents a potentially bewildering array of data and controls. Part of the design plan involved the methodical use of color to:

- Indicate classes of displays and controls.
- Offer cues to meaning (red/stop, green/go).
- Provide directions for use (color key controls to objects).

- Indicate state or when an object is selected.
- Distinguish among similar objects for ease of location.
- Please the eye and reduce strain.

2.5 The Value of Information at a Distance

Networks and telescopes exemplify the value of instruments which allow us to perceive information at a distance. We expected to benefit from the ability to monitor activity on a host elsewhere on the internet.

2.6 The Value of Information From the Past

The familiar aircraft flight recorder provided the concept for a handy function: the ability to record performance telemetry and replay it for later display. The displays and their controls would behave identically to a 'live' session. The household VCR was chosen as the model for KATO's flight recorder control panel.

2.7 Some Tough Decisions

A couple of issues caused some worry in the design phase. They will be introduced here and reviewed below in section 4.

An early decision was to connect KATO to the target via LAN. It remained to select a protocol and connection model. TCP was discarded due to its overhead on the target and the lack of anything reasonable to do when flow control occurred. UDP was selected because a streamlined implementation could easily be made, bypassing the relatively inefficient System V streams protocol stack. The KATO/UDP would plug directly into the LAN driver allowing us to telemeter streams protocol events without feedback effects. Packets dropped due to collisions and slow clients were of some concern. It was decided that the display client would be designed to tolerate lost packets.

The next issue was the choice of a connection model (which UDP does not provide). A minimalist approach was selected whereby:

- Connections would be end-to-end, one outstanding at a time.
- A connect request would turn on telemetry to the client, a disconnect request would turn it off.
- The above requests would be serviced at any time.
- Requests would be sent redundantly and serviced regardless of target connection state.
- Connect/disconnect requests would not be acknowledged.

The guidelines were simplicity, resource economy, and accommodating unreliable data delivery. Note that the target could be stolen by a connect request from another client. This drawback is offset by easy recovery from lost requests.

Security became a problem when remote tuning was included in the design. The capability to modify system tunable parameters from the remote client via UDP would allow anyone to connect and impair or crash a target. The solution was to hitch a ride on existing system security by requiring that a new utility remtuneok be run by root on the target to explicitly permit remote tuning (and other related activities).

3. KATO Today

Keeping in mind the design issues covered in the previous section, we turn now to a description of the tool in its current incarnation.

3.1 Alternative Methods of Displaying State Information

This section discusses the display types used in KATO. In section 3.2, we shall show how these displays are applied in the per-topic windows.

Most of the performance tools discussed above present data in numeric form. Those which utilize graphics provide only crude line graphs which jump-scroll left when the graph becomes full. KATO replaces these with a continuously left-scrolling chart recorder (Fig. 1) reminiscent of medical displays.

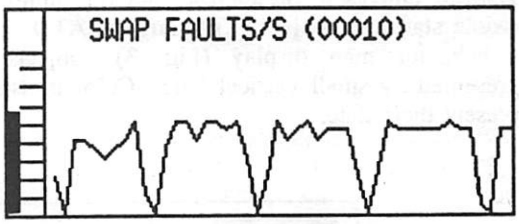


Figure 1: Chart recorder display

The parenthetical number by the title denotes the value of each interval of the dynamic scale at left. The minimax bar (left of scale) records the extremes of values which have passed through the display. Rendering cost is modest as polylines are undrawn in the background color, rather than using area operations to scroll the graph. Chart recorder displays are heavily used in KATO because patterns of behavior and correlations among them are often significant.

Four other display types were devised for KATO, each suited to a particular class of event. These are the n-ary event indicator, the bargraph, the state timeline, and the set behavior map.

Some events of interest have a single attribute which takes on one of a small set of logical values or states. In KATO, these events are displayed with n-ary event indicators, small text boxes that change border color to indicate state transitions through the set.

The bargraph style of display is used when a value with respect to some limit is of interest or when a comparison of similar parameters is desired, and when history of behavior is not important. The number at the bottom is the value of the parameter.

When a history of state changes is needed, the state timeline graph is used (Fig. 2). This is a continuously left-scrolling display in which each horizontal line records a particular state. State timelines are well suited to indicating the behavior of processes, devices, and communications protocols. The rendering cost is moderately high and proportional to [area X update-rate] due to the use of area copies for scrolling.

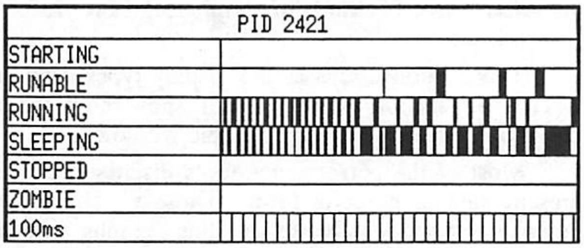


Figure 2: State timeline graph display

When the state of a large number of linearly organized objects is of interest, and the number of possible states per object is not large, KATO uses a set behavior map display (Fig. 3). Objects are represented by small vertical bars. Color is used to represent their state.

3.2 Window Organization

This section discuses the division of information into per-topic windows and the application of suitable display types to specific data items.

KATO initially displays a master window (Fig. 4), which provides top-level controls and status information. Bargraphs show the UDP bandwidth and percent real time consumed by the X client. N-ary event indicators tell of lost packets, connection status, and date/time. The master window contains a flight recorder control panel which follows a VCR model of interaction. A slidebar valuator shows seconds into the recording and provides a means of seeking to a particular offset. To the right, a menu of subwindows allows selection of target information by topic.

At the top line, buttons are provided to freeze and thaw all displays. The per topic windows described below all have localized versions of the same buttons.

The first of these topics is per process information (Fig 5). N-ary event indicators are dynamically created and destroyed to reflect the corresponding process activity. Processes are listed only by PID, a slight inconvenience tolerated in the interest of conserving resources. Green borders appear momentarily as processes acquire and lose the CPU. Process indicators can be selected and a state timeline display (Fig. 2) can be requested by mouse, giving more detail regarding scheduling behavior of selected processes.

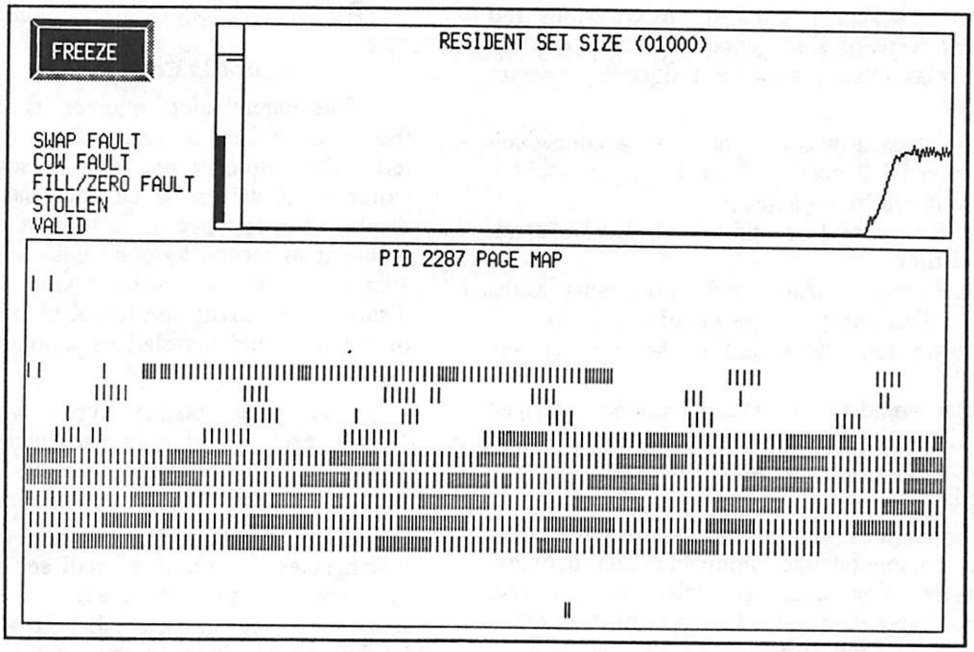


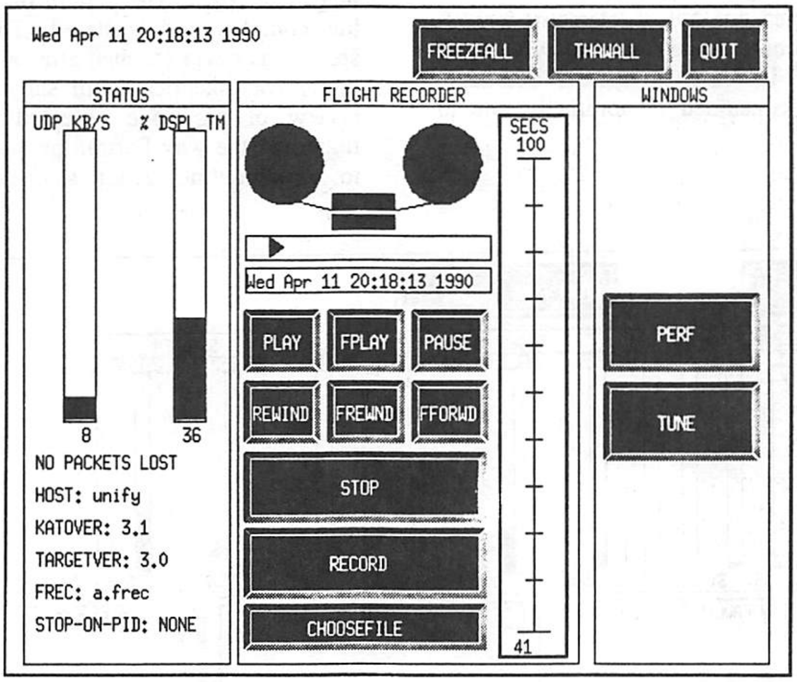
Figure 3: Set behavior map display

The memory window (Fig. 6) presents information on memory resource levels and usage, with the emphasis on paging activity. Bargraphs in the upper left break down free memory by 88K cache units(CMMUs). To the right, other bargraphs break down page aging and reclamation by age category. Chart recorders below monitor various types of faults, page stealing, disk traffic due to paging, and levels of several memory statistics crucial to OS allocation policies. A button at the top will pop a virtual page map window (Fig. 3) for the currently selected process.

The page map monitors changes in the state of virtual pages belonging to a process. Color is used to indicate when faults occur, when pages become valid, and when pages are reclaimed by vhand. The map is useful for analyzing the behavior of large virtual memory-bound programs in terms of minimizing page boundary crossings, stride sensitivity, and wasted space. Patterns of memory access can also provide clues to the application developer who knows something about the process's memory layout and usage.

The performance window (Fig. 7) displays information which will be familiar to users of sar. It consists of chart recorders monitoring the breakdown of CPU cycles, process scheduling, system call frequency, and IO activity.

The remote tuning window (Fig. 8) provides the capability to remotely view and edit the system table of dynamically tunable parameters. Buttons select parameters of interest and request reading/writing of the parameter set. The value of the currently selected parameter is displayed in several radixes at the bottom. A slidebar valuator at right is used to modify the current value. A typical usage is to run a test case on the target, and optimize parameters with real time feedback on the consequences of each change.

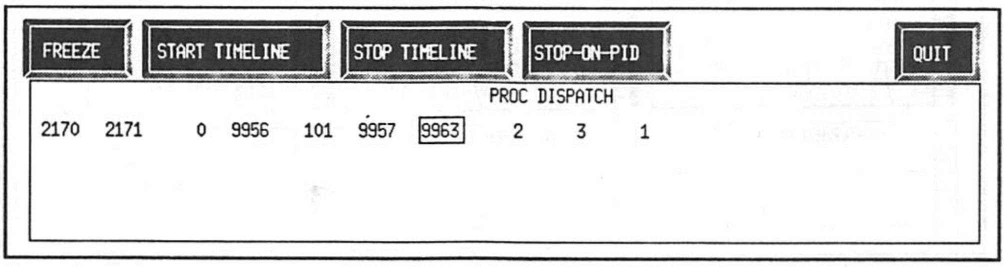


Wed Apr 11 20:18:13 1990

Wed Apr 11 20:18:13 1990

NO PACKETS LOST
HOST: unify
KATOVER: 3.1
TARGETVER: 3.0
FREC: a.frec
STOP-ON-PID: NONE

Figure 4: Master window



```
2170  2171    0  9956   101  9957  9963   2   3   1
```

Figure 5: Process window

4. Looking Back

At the time this is written, we have accumulated ten months of experience with KATO. This section evaluates its costs and benefits.

4.1 Development effort

KATO was designed and developed by the author over a six-month period, equivalent to about two and a half months of continuous effort. Two weeks of that time was spent in design. Expertise in X Window graphics, network protocols, System V internals, and performance analysis was required. Fig. 9 shows a breakdown by lines of the coding effort.

4.2 Impact on the OS

KATO was a moderately invasive tool, requiring telemetry calls to be inserted into 19 existing OS functions. As such, it will contribute to the difficulty of further development, and of incorporating new OS releases. This is the cost of acquiring broad based, low overhead performance data at a high time resolution.

Three kilobytes were added to the OS size, most of which occupied one file of telemetry formatting and buffering code. Measurements with SAR show no noticeable increase in system overhead. Network bandwidth consumed is nominally one to five kilobytes/second.

4.3 Three Case Histories

Several examples drawn from our experience with KATO illustrate its application to practical problems. The first example demonstrates the solution of an application performance bug, complicated by an inefficient system paging algorithm. The second recounts the detection of stride sensitivity in a memory management policy. The third is an example of how KATO is used for optimizing system tunable parameters.

CASE HISTORY 1. A customer reported that a new CFD application being used as a benchmark was running slowly. The application should have been compute bound under ideal conditions. Fig. 10 below is a snapshot of KATO applied to this problem. User cycles and free memory are down, and the system is paging at it's maximum rate. The application engineer claimed the program was carefully ordered to minimize page boundary crossings and should run in four megabytes (this on an eight megabyte configuration). Looking at a virtual page map of the process, we saw page faults occurring over a 12 megabyte space. The pattern of accesses was orderly but complex and scattered. Three quarters of the space was never touched after initialization. The customer was skeptical until shown the display, but a review of his code revealed a false assumption regarding the way Fortran passes variable-size arrays to a subroutine. After a one-line change to the

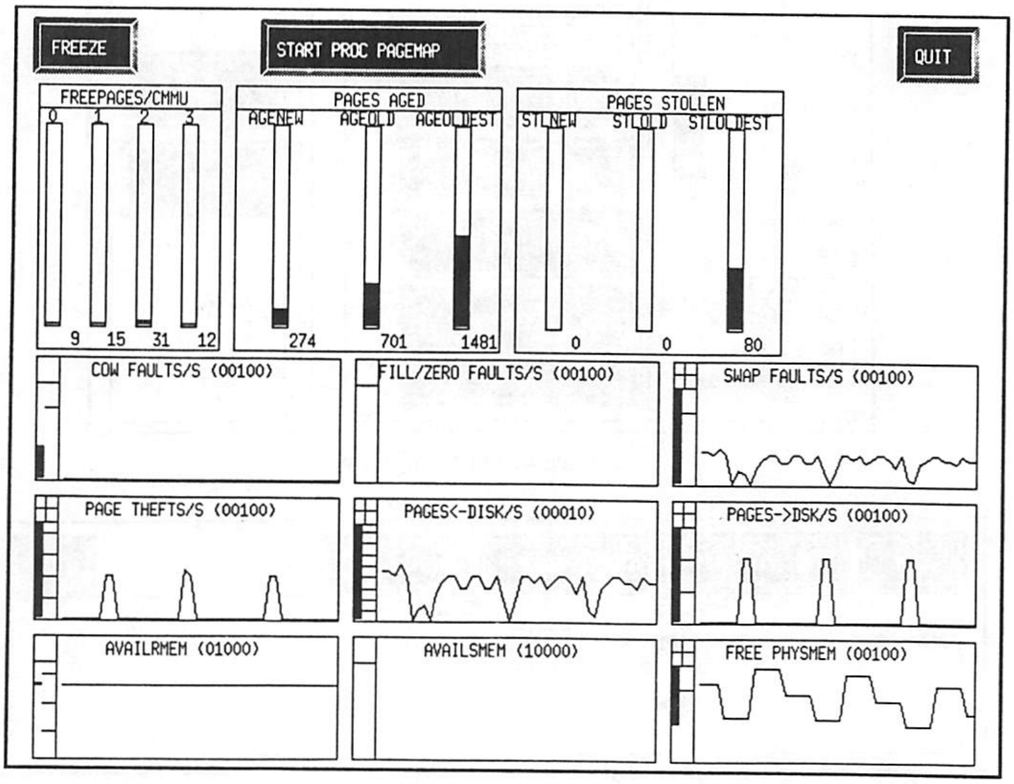


Figure 6: Memory window

program, it ran ten times faster on the other vendor's machines, but only five times faster on ours. We took the same user cycles and the same number of faults to run the job. A review of our code revealed that we had forgotten to implement a common optimization to outbound paging, which when installed, brought us up to speed.

CASE HISTORY 2. A government agency provided a 2D FFT benchmark which ran two orders of magnitude more slowly than on other machines with similar memory size and slower CPUs. KATO's memory window and page map told the story at a glance (Fig. 11). The page map showed very orderly accesses to virtual addresses at 16K intervals. The free pages per CMMU display showed one of four

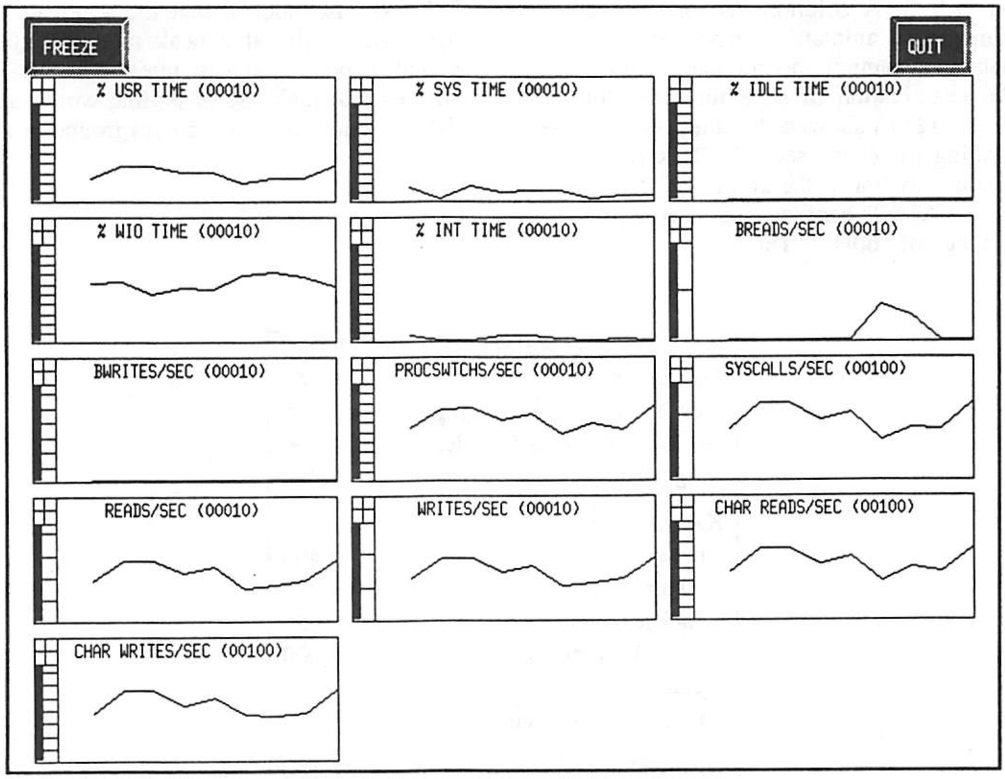
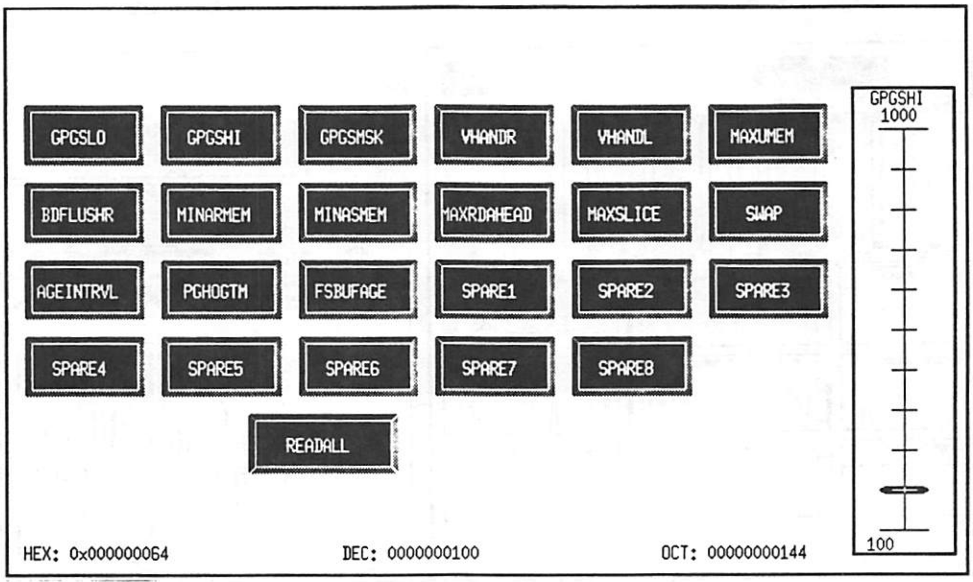


Figure 7: Performance window



HEX: 0x00000064 DEC: 0000000100 OCT: 00000000144

Figure 8: Remote tuning a window

CMMU partitions empty, and the others full. As the benchmark progressed, the imbalance would rotate through the partitions while the 16K access pattern would rotate through the pages. Stride sensitivity in memory controllers is a well known phenomenon. We solved it in this case by dynamic migration of physical pages between partitions, after which the benchmark ran 100 times faster.

CASE HISTORY 3. A solution vendor complained that their foreground animation display task moved in bursts while a compute-bound background task was run. An examination of state timelines for the two tasks (Fig. 12a,b) showed that they fully utilize the CPU, trading off every second. This accounted for a one-second rhythm in the animation. System V as provided by AT&T configures a maximum process time slice of 60Hz. This may have been appropriate for low performance machines of twenty years ago with their high context switching overhead, but not for today's workstations. Using the remote tuning window, the MAXSLICE parameter was reduced to 5Hz (83ms). The timeline displays changed to Fig. 12c,d and the animation became continuous. System overhead and execution times of the two tasks were unaffected, as one might expect.

We also noticed that the application was making system calls at a peak rate of 12,000/sec as it polled a busy graphics interface. The vendor was advised that their use of polling would nearly double the execution time of the background task.

Target Kernel	Lines
New io/kato.c	700
New sys/kato.h, kato_events.h	730
Added to existing OS files	50
Target Kernal total	1,480
KATO client:	
network	400
display widgets	5,300
packet parsing	600
topical windows	3,000
gluer	400
KATO client total	9,700
GRAND TOTAL	11,180

Figure 9: KATO code breakdown including white and comment lines

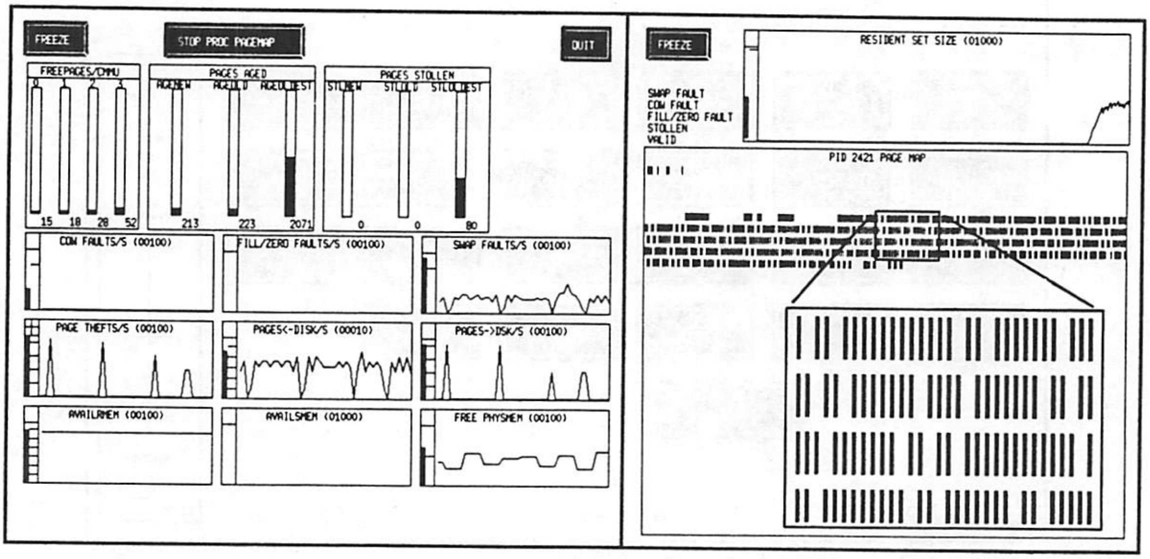


Figure 10: Application using an unintentionally sparse array

4.4 Things That Went Wrong

A comparison of design decisions discussed in section 2 against experience with the implementation yields several weaknesses worthy of mention.

The connection model (2.7) does not accommodate more than one client. This can be a source of mystery when someone connects a second instance of KATO to the same target and all telemetry is silently redirected to the new client.

As kernel development proceeded in parallel with KATO, synchronization of client vs. target data structure formats became a problem only partially resolved by addition of version stamps to the telemetry. In particular, the verbatim transmission of kernel structures was a mistake. It would have been better to copy the data into a more stable canonical format.

It turned out that there were several circumstances in which X could block a client indefinitely (e.g., while the window manager grabbed the mouse to prompt for window placement). This occasionally resulted in dropped packets and temporarily perturbed displays.

5. KATO Tomorrow

At this time (March 1990), additional functionality is still being added. Several new topical windows are being developed to display disk and buffer cache behavior, per device read/write/ioctl rates, system call statistics, streams activity, and dynamic kernel profiling.

The wealth of performance-related data gathered here provides the raw material for a truly effective configuration tuning advisor (a la tunex, section 1.2). The trick, of course, will be embedding the expertise to evaluate the data and construct effective advice. The feedback-driven behavior of many kernel algorithms and their mutual interactions is expected to present a significant challenge.

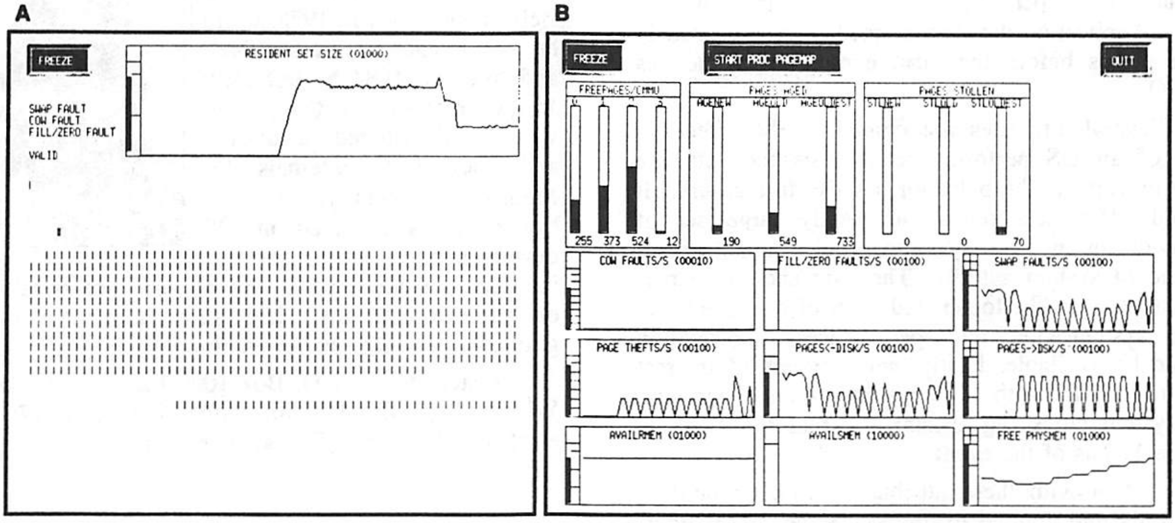


Figure 11: A stride sensitive memory controller policy

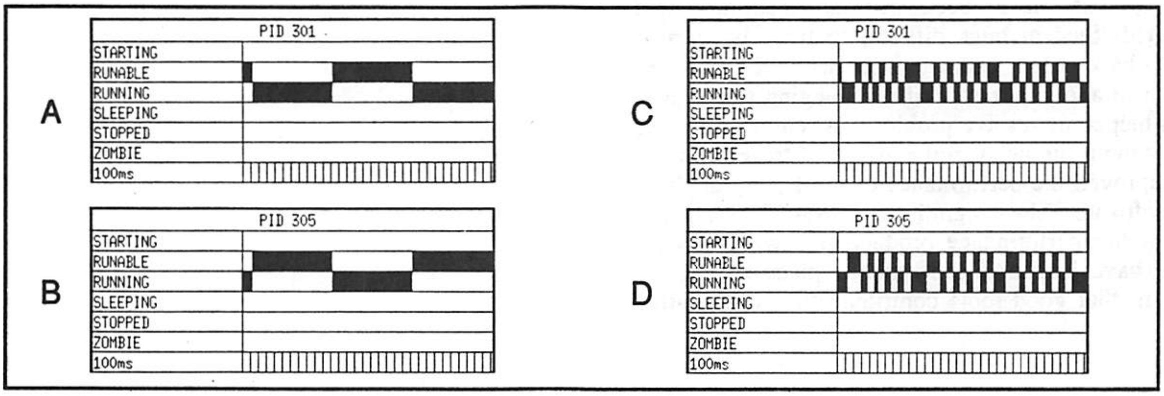


Figure 12: (a,b) Animation and background task state timelines with MAXSLICE = 60Hz. (c,d) The same with MAXSLICE = 5Hz.

The current invasive approach to data gathering precludes applying KATO without access to kernel source. A moderately portable daemon running on the target could acquire data via /dev/kmem without modifying OS code. The cost would be lower time resolution for some displays, loss of some information, and a significant impact on target cycles.

6. Conclusions

Our experience has shown that both system and application developers can benefit significantly from performance analysis tools with finer time resolution and broader scope than those currently available. There is a critical mass of data presentation (in terms of scope and clarity) which conveys a gestalt of OS/application interactions. This is an example of what we call the "threshold of visualization."

Raw statistics are useless without skilled interpretation. Anyone may quickly learn to operate kato, sar, prof, etc., but little can be accomplished without expertise in performance issues such as the impact on an operating system of demands for services. Application developers need to be educated in these issues before they can exploit tools such as KATO.

Several attributes are crucial to the effectiveness of an OS performance analysis tool. Its use must not impact the behavior it seeks to measure. It should offer data on a sufficiently large set of behaviors in such a way as to provide an integrated picture of system activity. The user should be permitted to select the topics and level of detail desired. Both system-wide and per process information should be available. Lastly, some events of interest require display with time resolution on a par with the system clock (eg. 16ms) and real time display within 100ms of the event.

A tool with these attributes has been built at Tektronix and applied to the XD88 family of superworkstations. KATO has more than justified its initial development cost in a number of ways. Performance leaks have been rapidly identified and corrected. System bugs difficult to trace by normal means have been characterized with KATO well enough to apply more specific debugging techniques. It has helped us resolve problems at remote customer sites without the delay and expense of travel. Its use has improved the performance of third party application software. Most significantly, KATO has given us a higher performance product than we otherwise would have had, and confirmed a piece of popular wisdom: that good tools contribute to a competitive edge.

7. Acknowledgements

This work would not have been possible without the generous support of many friends. Tony Birnseth started me thinking about performance tools, and provided valuable criticism and useful ideas. Jack Sterett, Ross Ryding, and Wayne Matterson lent legitimacy and official support to the project. Dan Hobbs ruthless reviews of this paper will no doubt save me major embarrassment.

8. REFERENCES:

R. Rodriguez, *A System Call Tracer for UNIX*, USENIX Conference Proceedings, pages 72-80 (June, 1986).

S. Zhou, H. Da Costa, and A. Smith, A File System Tracing Package for Berkeley UNIX, USENIX Conference Proceedings, pages 407-419 (June, 1985).

Roy Trammell discovered UNIX when he joined AT&T Bell Laboratories in 1978 where he did violence to releases three and five. In 1984 he left New Jersey, moving to Oregon to write a distributed filesystem and hack BSD internals for Metheus. Currently, at Tektronix, he is involved in OS development, graphics interfaces, network code, and performance tweaking on superworkstations. His interests include fine-grained parallelism and animation.

Reach Roy at P.O. Box 1000 Mail Sta. 61-183; Wilsonville, OR 97070; phone 503-685-2297; or electronically at royt@orca.wv.tek.com.

USENIX Summer Conference
June 11-15, 1990
Anaheim, California

Perspectives on NFS File Server Performance Characterization

Bruce E. Keith – Digital Equipment Corporation

ABSTRACT

Two major approaches to Network File System (NFS) file server performance characterization exist today. One approach, denoted the "synthetic-workload, single-client" (SWS) approach, uses an NFS workload abstraction in terms of an NFS operation request mix and an NFS operation request rate as input to a load generator utility running on a single, or small number of NFS clients [LEGATO89], [SHEIN89]. Another approach, used within Digital Equipment Corporation and denoted the "actual-workload, multiple-client" (AWM) approach, is to execute an actual workload on multiple NFS clients. In both approaches, various performance parameters are monitored while an NFS load is applied to the server.

This paper discusses the results of an initial evaluation of the SWS approach's ability to generate server and network loads and associated client response that are equivalent to those generated by the AWM approach. The paper further discusses the fundamental reason for investigating file server performance: helping a computing facility answer the question "How will our application (workload) perform using this file server?"

1. INTRODUCTION

Network File System (NFS) file servers are key components in today's distributed computing environments. During a computing facility's NFS file server selection process, the individuals making the selection decision need to know how a given NFS file server configuration performs so that comparison can be made among servers from different vendors. Ultimately, the individuals need to know how well the computing facility's particular application, or workload, will perform using a given server configuration.

There have been two major approaches to NFS file server performance characterization to date. One approach, denoted the "synthetic-workload, single-client" (SWS) approach, requires a computing facility to develop an ad hoc benchmark that is considered representative of the computing facility's workload (application). The benchmark is then executed on a single (or small number of) client(s) of the NFS file server under test, which in turn impose(s) a load on the file server under test. Thus, the single client emits NFS requests as if it were a larger number of clients.

Another approach, denoted the "actual-workload, multiple-client" (AWM) approach, involves the simultaneous execution of an actual workload on multiple clients of the NFS file server under test, which in turn impose a load on the file server supporting the clients.

In both approaches, client and server performance are measured while the load is applied to the server, with performance typically expressed in terms of the clients' resultant response time and throughput and the server's corresponding resource utilization.

Refinements to the SWS approach have been made recently by [LEGATO89] and [SHEIN89]. These refinements are:

1. Abstraction of a file server's workload in terms of its NFS operation mix and NFS operation request rate.
2. Development of utilities that can generate an NFS load based on input expressed in terms of this workload abstraction.

This paper discusses the results of an initial evaluation of the refined SWS approach to NFS file server performance characterization. Specifically, the paper discusses the ability of the refined SWS approach, as instantiated by the nhfsstone NFS load generating program [LEGATO89], to generate server loads and client response that are equivalent to those produced by the AWM approach used within Digital Equipment Corporation over the last 3 years.

1.1 NFS Functionality

NFS provides the ability to share file systems transparently in a heterogeneous environment of processors, operating systems, and networks. Sharing is accomplished through a cooperative mechanism in which a computer system, denoted a "server," exports (or offers) some or all of its file systems to other computer systems on a network. A "client" is any other computer system on the network that references any of the file systems exported by the server. A given computer system may act as either a server, a client, or both.

Once a client has remotely mounted a file system exported by the server, the client can then access the server's file system as if it were locally available on the client. This eliminates the need to copy files from one system to another before a client can reference a file. Consequently, only one copy of a file need be maintained on the server rather than several copies of the file on several different computer systems.

Internally, an NFS client makes requests of the server for files within an exported file system through the NFS protocol. The NFS protocol defines specific types of operations that a client system may request of a server [SANDBERG85].

1.2 SWS Approach

The types of operations defined by the NFS protocol provide the key to one of the significant refinements made by [LEGATO89] and [SHEIN89] to the ad hoc benchmark, SWS approach.

Both [LEGATO89] and [SHEIN89] abstract an NFS file server's workload in terms of a mix of the types of NFS operations expressed as a percentage of the total number of requests made by all of the clients supported by the server. The abstraction is further extended by both authors to include the rate at which the NFS clients make requests of the server. Thus, [LEGATO89] and [SHEIN89] suggest that an NFS server's actual workload can be abstracted in terms of an NFS operation mix and an NFS operation request rate. Both NFS operation mix and NFS operation request rate are quantities that can be readily determined through the nfsstat NFS statistics reporting utility.

The second significant refinement from [LEGATO89] and [SHEIN89] is the development of utilities to generate an NFS load based on the NFS workload abstraction of an NFS operation mix and an NFS operation request rate. The utilities are executed on an NFS client to generate a load on an NFS server.

The nhfsstone utility is a suitable choice for an NFS load generating tool. Thus, the nhfsstone NFS load generating utility developed by Legato Systems, Inc. was used as the SWS approach to NFS file server performance characterization within this evaluation.

The nhfsstone utility requires the following input parameters:

- An NFS operation mix file containing the results of an execution of the nfsstat NFS statistics reporting utility.
- An NFS operation request rate.
- A list of remote file systems to serve as targets for the generated NFS requests.
- Either the length of time for which the utility is to generate the synthetic load or the total number of NFS requests that are to be generated by the utility.

Upon completion of execution, the nhfsstone utility reports the actual NFS operation mix generated, the elapsed execution time, the total number of NFS operation requests generated, the resultant NFS operation request rate, and the average service time for all of the generated NFS operation requests. Consult [LEGATO89] for a description of additional output generated by the utility.

Thus, the SWS approach characterizes NFS file server performance in terms of NFS request service times and NFS request throughput experienced on an NFS client.

1.3 AWM Approach

The AWM approach to NFS file server performance characterization used within Digital Equipment Corporation during the past 3 years applies a real user-level task workload to ULTRIX NFS diskless client workstations, which in turn apply a load to the NFS server. As many as 50 ULTRIX NFS diskless client workstations have been configured on a private Ethernet and supported by a single ULTRIX NFS file server.

As the workload is applied to progressively larger numbers of client workstations, user-level task service time and throughput are measured on the client workstations. Server resource utilization is simultaneously measured in terms of CPU utilization and disk and network interface I/O operation rates. Further, network utilization is simultaneously measured on the Ethernet.

The workload applied to the client workstations emulates a software engineering environment in which a single software engineer is assigned to each NFS client workstation. Each client workstation executes a unique, repeatable sequence of user-level tasks (commands) selected from a common pool of the most frequently executed user-level tasks encountered in a software engineering environment. Thus, the workload is not a lock step workload in which each workstation executes the same task at the same time. Rather, the workload follows typical usage patterns in which individual users collectively execute a variety of user-level tasks at a given point in time.

Hence, the AWM approach characterizes NFS file server performance in terms of user-level task service times and throughput experienced on NFS clients. The AWM approach also characterizes NFS file server performance in terms of the associated utilization of server and network resources required to provide a particular degree of user-level task response.

2. EVALUATION METHODOLOGY

2.1 Focus

The focus of the initial evaluation of the SWS approach as implemented by the nhfsstone NFS load generator utility was to determine nhfsstone's ability to duplicate NFS file server utilization, network utilization, and client response levels previously measured by the AWM approach. Further, the evaluation was to determine if the SWS approach could be used as a substitute for the AWM approach as an NFS server sizing method. The evaluation was not intended to be a critique of the nhfsstone load generator utility itself.

The following primary questions were addressed by the evaluation:

1. Is the NFS operation mix of an actual workload invariant with respect to scaleability? Specifically, is the NFS operation mix generated by a single NFS client equivalent to the NFS operation mix generated by ''N'' clients? This is the fundamental assumption of the SWS approach that provides the basis for using a single client to simulate ''N'' actual clients.
2. Does the NFS operation request rate of an actual workload increase linearly with respect to scaleability? Specifically, is the NFS operation request rate of ''N'' clients equivalent to the average NFS operation request rate of one client multiplied by ''N''? This point addresses the issue of how a server responds as its load (client NFS operation request rate) increases.
3. Is the workload offered by the nhfsstone load generator equivalent, in terms of server and network utilization, to the offered load generated by the AWM approach. Further, can the load offered by ''N'' actual clients be simulated by adjusting nhfsstone's NFS operation request rate while holding the operation mix constant?
4. Do NFS operation service time trends, as reported by nhfsstone, have a relationship to the user-level task service time trends measured by the AWM approach?
5. What issues, if any, exist concerning the usage of the nhfsstone load generator utility?

2.2 Strategy

The evaluation strategy was to configure an NFS file server that had been previously characterized by the AWM approach. The client NFS operation mix and operation request rates measured during the previous characterization of the server were used as input to nhfsstone. Server and network resource utilization were measured while nhfsstone applied the load to the server for a 1-hour interval. The client NFS operation mix and operation service times reported by nhfsstone were recorded and verified with parallel measurements made by the nfsstat NFS statistics reporting utility on the

Performance Monitoring Tool	Metric
vmstat	Average server CPU idle time Simultaneous average and maximum server disk I/O operatoin rates
netstat	Average and maximum server network interface packet rates
nfsstat	NFS operation reqeust mix and rate
nhfsstone	Average NFS request service time
LAN Traffic Monitor	Ethernet network utlization

Table 1: Metrics and Tools

	AWM/SWS Approach File Server	AWM Approach Diskless Clients	SWS Approach *nhfsstone* Platform
System	DECsystem 3100	DECstation 3100	DECstation 3100
Memory	24 Megabytes	16 Megabytes	16 Megabytes
Disks	3 RZS5 (332 MB SCSI)	N/A	1 RZS5
Software	ULTRIX V3.1 (RISC)	ULTRIX Worksystem Software V2.1 (RISC)	ULTRIX Worksystem Software V2.1 (RISC)

Table 2: Testbed

nhfsstone platform.

2.3 Metrics and Tools

Wherever possible, standard ULTRIX and UNIX system performance monitoring utilities were used to monitor server and nhfsstone platform performance. Table 1 summarizes the performance metrics and associated performance monitoring utilities used during the evaluation.

2.4 Testbed

Table 2 summarizes the hardware and software used during the evaluation. A private Ethernet network was used to interconnect the NFS server and clients. Note that a local disk was used for the ULTRIX operating system on the nhfsstone platform so as not to place the load of an additional diskless client on the server during testing.

3. RESULTS

3.1 Summary of Findings

The following list summarizes the findings of the evaluation, with further detail in the following sections.

1. NFS operation mix was found to be invariant with respect to scaleability. Thus, the assumption that a single client can simulate 'N' actual clients in terms of a constant NFS operation mix was validated. Additionally, nhfsstone accurately reproduced the requested operation mix.
2. NFS client operation request rate was found to be non-linear with respect to scaleability, due to server response degradation. This implies that a rate other than "N" times the average NFS operation rate of one client must be specified as input to nhfsstone for each level of clients (load) in order to simulate the actual load. This prohibits nhfsstone from being a total replacement for the AWM approach.
3. Average server and network resource utilization levels were equivalent between the AWM approach and the SWS approach, however maximum resource utilization levels were significantly lower with the SWS approach than with the AWM approach.
4. A relationship between the NFS operation service times of the SWS approach and the user-level task service times of the AWM approach was found. Thus, nhfsstone results can serve as a rough, ballpark indicator of client application performance.
5. No major configuration or capacity issues arose concerning the nhfsstone platform throughout the evaluation.

3.2 Invariance of NFS Operation Mix

NFS operation mix was found to be invariant with respect to scaleability by inspection of measured data obtained through the AWM approach. The NFS operation mix varied no more than two percent across the range of clients. Thus, the fundamental assumption of the SWS approach that a single synthetic client can simulate "N" actual clients in terms of NFS operation mix for a given actual work-

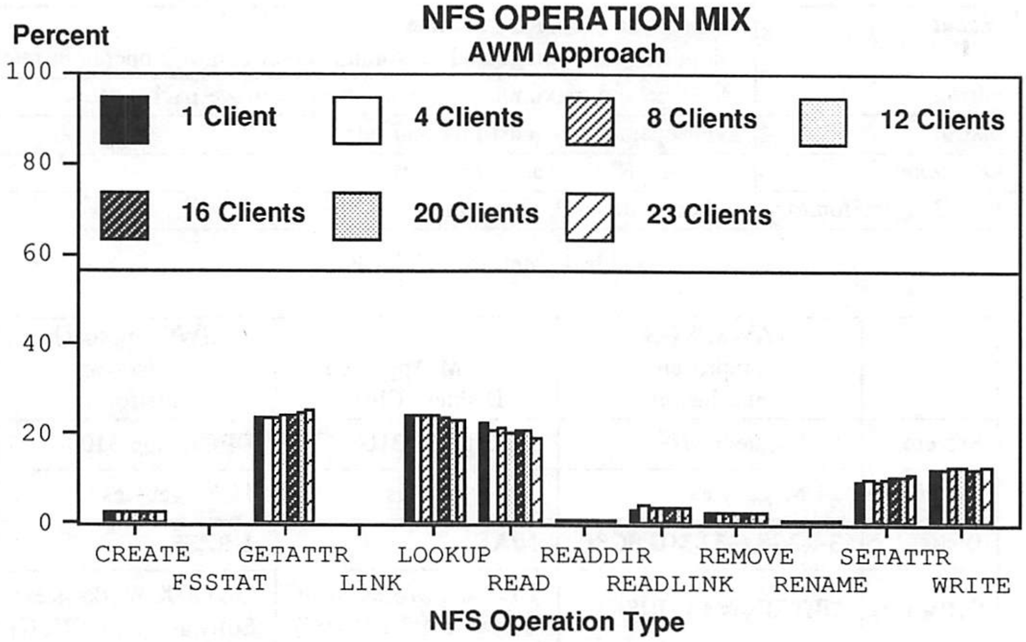


Figure 1: NFS Operation Mix

load was validated.

Invariance of NFS operation mix is an important property since a server's response will vary as NFS operation mix is varied. Since NFS file server performance characterization is primarily interested in the server's response to increased load for a given NFS operation mix (workload), any change in server response due to variance of NFS operation mix would skew the desired characterization results.

A server's response changes as NFS operation mix is varied since not all NFS operation types are equally expensive in terms of the work that the

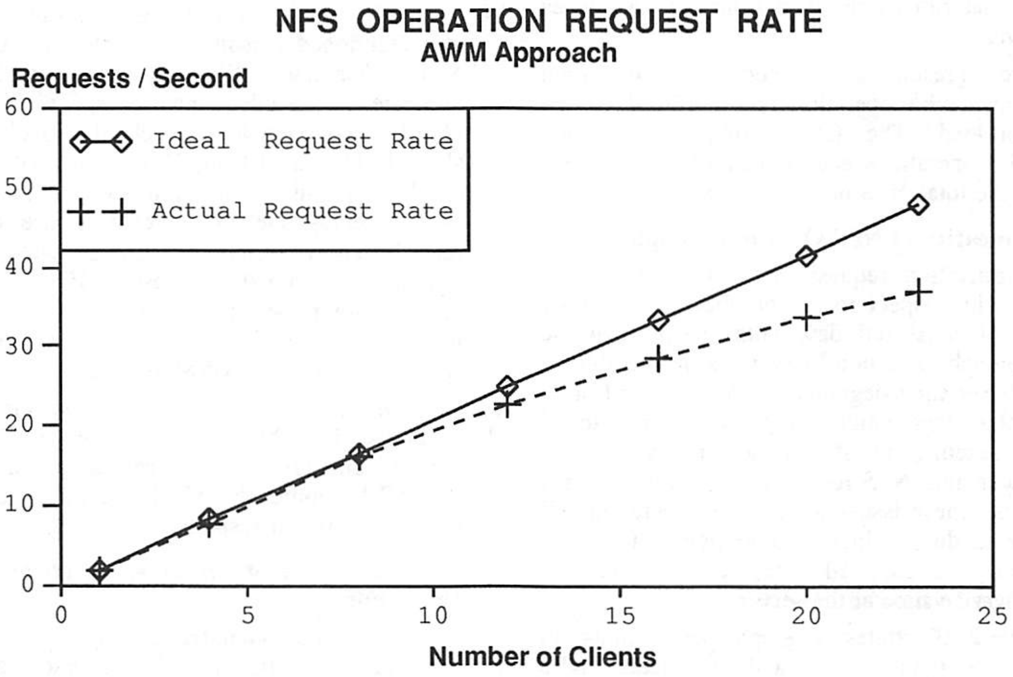


Figure 2: NFS Operation Request Rate

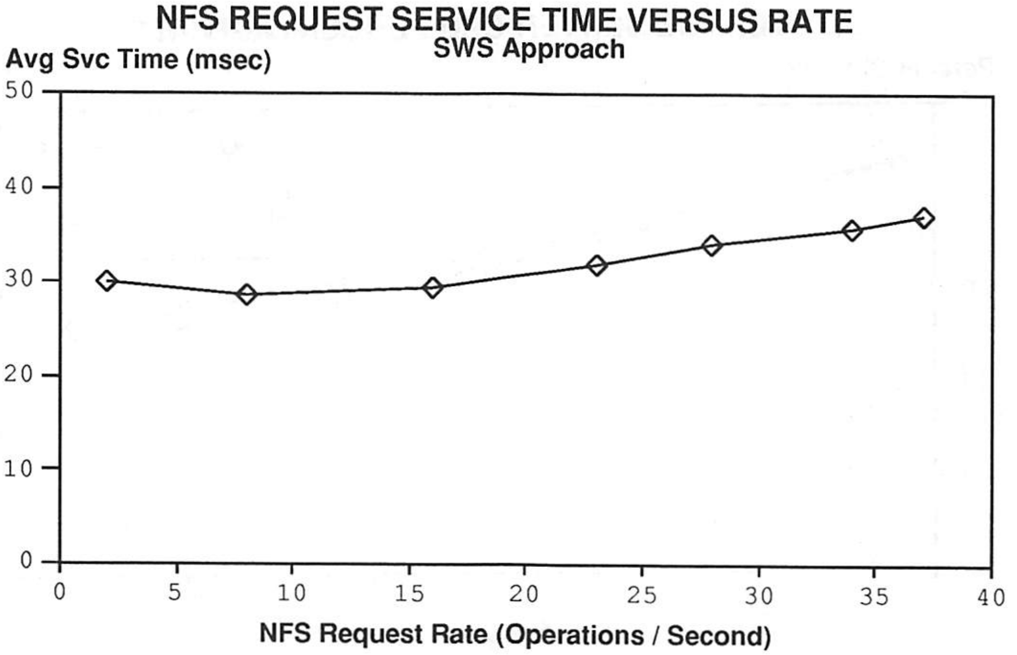


Figure 3: NFS Request Service Time

server must accomplish to process the NFS request [BRIGGS88]. The underlying issue is that, since an NFS server is stateless, the server must relegate any modified data to non-volatile storage before notifying the client that the requested operation was completed [SANDBERG85]. For example, [BRIGGS88] reports that NFS operations involving file system changes are much slower in terms of server response than operations that can be resolved from various caches on the server.

Figure 1 presents a bar graph of NFS operation mix for those NFS operation types utilized by the actual workload. The fsstat, link, readdir, and rename NFS operations each constituted less than 1 percent of the total NFS operation mix.

3.3 Non-linearity of NFS Operation Request Rate

NFS operation request rate was found to be non-linear with respect to the number of clients by inspection of measured data obtained through the AWM approach. The non-linearity is attributable to server response time degradation and a second-order effect of this degradation. In particular, the actual workload, reacting to slower server response, is unable to generate NFS requests as quickly. The net effect of both these issues is that NFS operation call rate increases due to increased numbers of clients, but individual client call rate decreases due to increased service time at the server.

Figure 2 illustrates a graph which plots an "ideal" NFS request rate and the actual NFS request rate versus the number of active NFS clients as measured through the AWM approach. The "ideal" NFS request rate is defined by the case where the NFS operation request rate of "N" clients is equivalent to the average NFS operation request rate of one client multiplied by "N". In this situation, an "ideal" or "responsive" server [KLEINROCK75] responds to increased client request rates with no performance degradation whatsoever.

The difference between the "ideal" and "actual" server curves is attributable to the aforementioned reasons. The difference implies that a rate other than "N" times the average NFS operation rate of one client must be specified as input to nhfsstone for each level of clients (load) in order to simulate the actual load. This means that degradation of NFS operation rate of the workload per unit of load (clients) due to server response degradation must be known empirically and supplied as input to nhfsstone through the requested NFS operation rate, in order for nhfsstone to duplicate the actual workload. This prohibits nhfsstone from being a total replacement for the AWM approach.

Figure 3 illustrates NFS request service time as measured by nhfsstone. The specified nhfsstone input parameters were set according to the results of the AWM approach, which implicitly contained a measure of server response degradation.

3.4 Equivalence of Server and Network Resource Utilization

Nhfsstone generated average server and network resource utilization levels that were comparable to those obtained through the AWM approach. The average load offered by "N" actual clients was

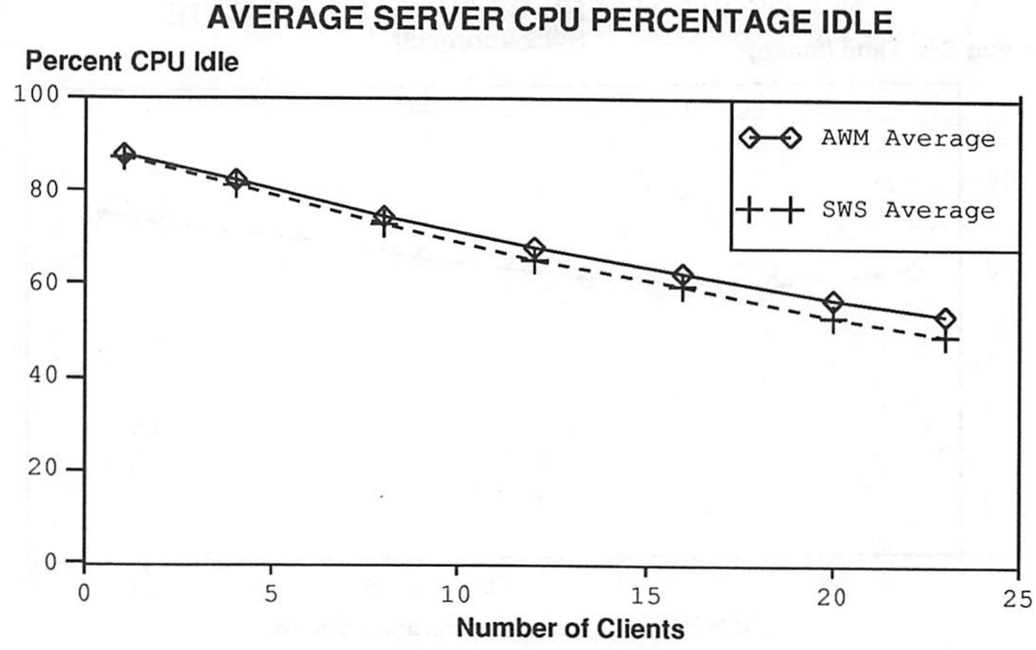


Figure 4: Average CPU Idle

successfully simulated by adjusting nhfsstone's NFS operation request rate while holding the NFS operation mix constant. The resultant average load and associated server and network utilization levels can be used to investigate server performance issues by holding NFS operation mix and request rate constant, varying the server configuration, and observing changes in NFS request response time.

However, nhfsstone did not produce maximum utilization levels that were comparable to those experienced with the AWM approach. The maximum utilization levels produced by nhfsstone were significantly lower than those experienced with the

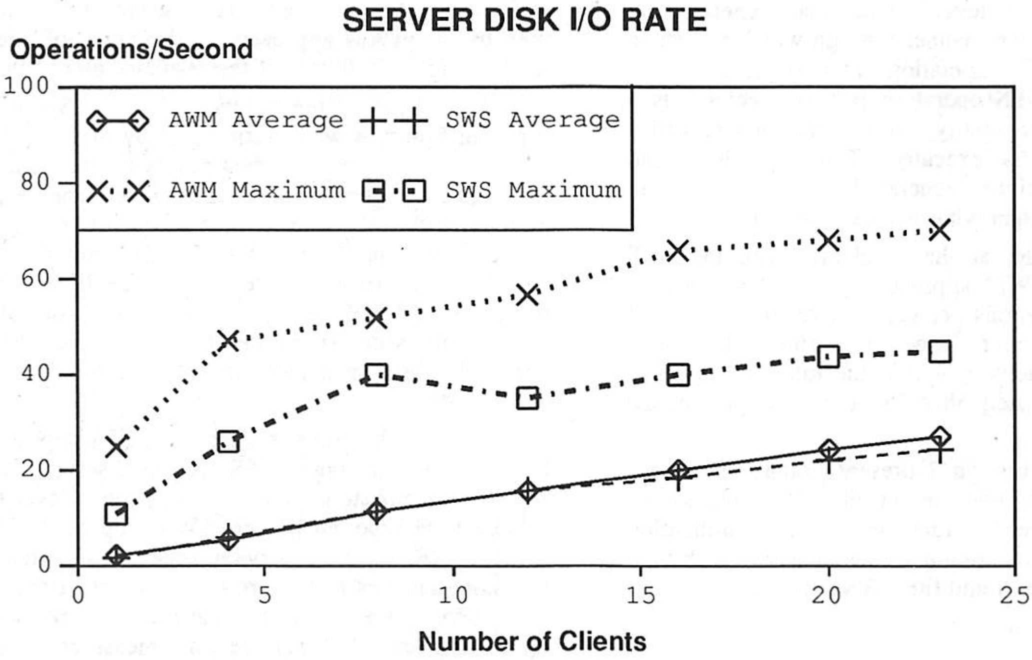


Figure 5: Server Disk I/O Rate

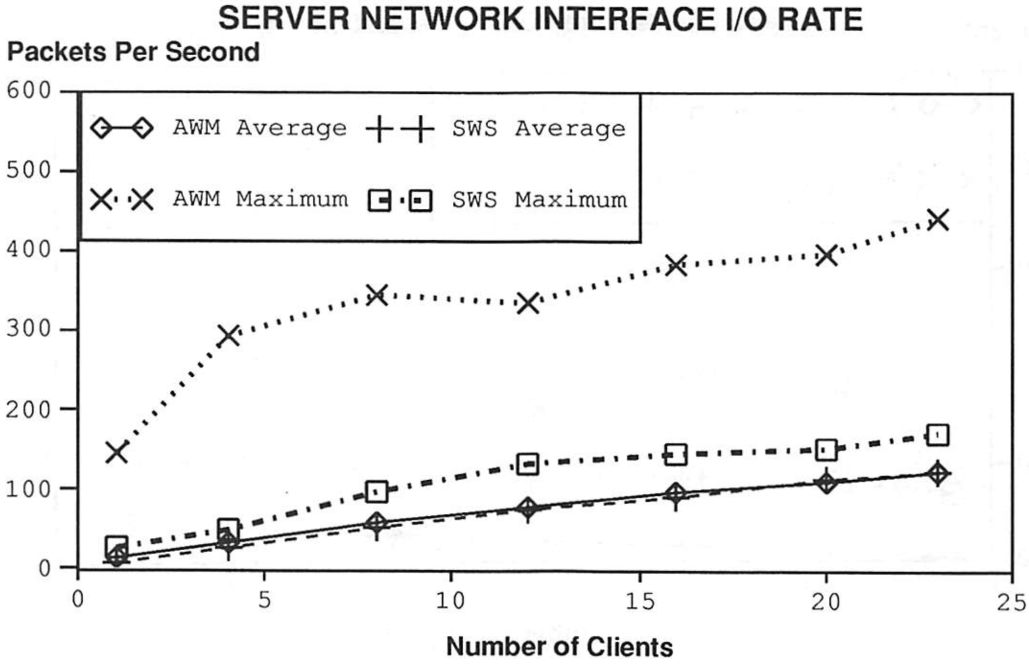


Figure 6: Server Network Interface I/O Rate

AWM approach. This can bias the average NFS request service times reported by nhfsstone in that the average service times reported can be lower (more optimistic) than what would be obtained with an actual workload. The reason is that maximum resource utilization levels increase NFS request service times.

The reason for the absence of maximum resource utilization levels in the load generated by nhfsstone lies in the manner through which nhfsstone allows the NFS operation request rate to be specified. The NFS operation rate is specified as a constant when, in reality, NFS operation rate varies as a workload is executed. Thus, the load and resource utilizations generated by nhfsstone are much smoother than what occurs in practice.

For example, at the 23-client level, the NFS clients in the AWM approach requested an average of 37 NFS operations per second over the entire test interval of one hour. However, during a 10-minute peak period of activity within the total test interval, the clients requested 58 NFS operations per second from the server.

Figures 4 through 7 present graphs of percentage server CPU idle, server disk I/O rate, server network interface I/O rate, and network utilization for average and maximum values achieved with both the AWM approach and the SWS approach.

3.5 Relationship of Client Response Trends

In order to serve as an indicator of client application performance, average NFS request service time reported by nhfsstone must relate to the average user-level task service time reported by the AWM approach. To investigate the presence of a relationship, the normalized average NFS request service time reported by nhfsstone was plotted with the normalized average user-level task service time measured by the AWM approach for the range of clients tested. Figure 8 illustrates the resultant graph.

The values of the points on each curve of the graph in Figure 8 were normalized by dividing each of the curves' respective data points by the value of the respective curve at the 1-client level. The desired effect was that the y-axis value of each curve at the 1-client level had a value of 1, to facilitate comparison. The measured average service time values of nhfsstone's NFS requests were on the order of 28 to 37 milliseconds while those of the AWM approach's user-level tasks were on the order of 8 to 12 seconds.

The graph suggests that a relationship exists between the average NFS request service time reported by nhfsstone and the average user-level task service time reported by the AWM approach. Thus, average NFS request service time as reported by nhfsstone appears to be a rough indicator of application performance, and in this case, a ballpark indicator of user-level task service time measured through the AWM approach.

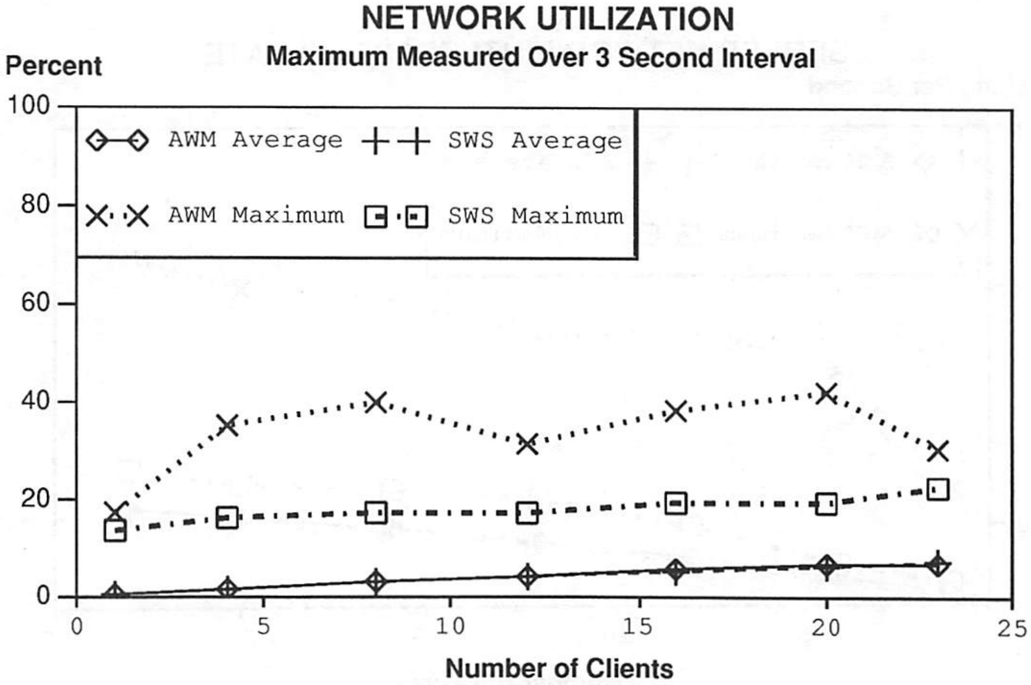


Figure 7: Network Utilization

It should be noted that NFS request service times as reported by nhfsstone form an upper bound on client NFS-related degradation since nhfsstone attempts to defeat data buffer, file attribute, and directory name lookup caches that exist on the client [SANDBERG85]. These caches help reduce the number of NFS requests that the client must issue for the client application to accomplish its work.

In the initial evaluation documented herein, the exact nature of the relationship between the average NFS request service time of nhfsstone and the average user-level task service time of the AWM approach could not be determined. One factor that influenced this outcome was that the average NFS request service time of nhfsstone and the average user-level task service time of the AWM approach are not independent variables, thus preventing the determination of a statistical correlation. Another factor was one of measurement precision in that NFS request response was measured on the order of milliseconds by nhfsstone while the AWM approach measured user-level task response on the order of hundredths of seconds. A test procedure that resolves both of these issues has been identified and will be included in future work.

The determination of the nature of the relationship serves as an ideal starting point for future work. Given that response trends are typically exponential, the relationship here could very well be exponential. Further, Figure 8 suggests that the relationship might be sinusoidal.

3.6 nhfsstone Platform Issues

No major issues were uncovered during the evaluation concerning the usage of nhfsstone. A 16-Megabyte DECstation 3100 was easily able to duplicate server and network utilization levels associated with 23 clients executing the actual workload and beyond. At no point during the evaluation was the platform unable to deliver the requested load using 12 nhfsstone subprocesses.

Server filesystem layouts and the number of server disks can be significant performance factors when configuring a server, however, this was not the case during this initial evaluation. For example, comparable results were obtained when nhfsstone referenced three file systems on the server as when four file systems were referenced on the server.

4. CONCLUSIONS

Nhfsstone produced average server and network utilization levels that were comparable to those experienced with the AWM approach. This proved the validity of the NFS workload abstraction in terms of an NFS operation mix and an NFS operation request rate. Further, nhfsstone accurately generated requested NFS operation mixes and NFS operation request rates during the evaluation.

The average NFS loads generated by nhfsstone can be used to investigate server performance. Changes in average NFS request service time can be observed while modifying the configuration of the server and holding NFS operation mix constant. Graphs similar to Figure 3 which plot NFS operation

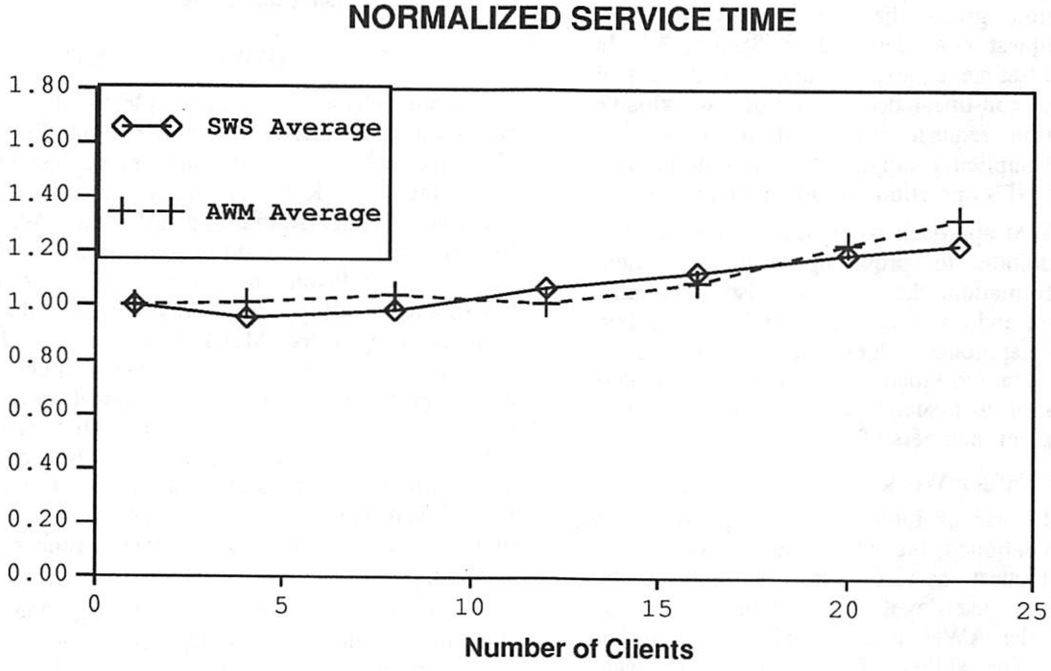


Figure 8: Normalized Service Time

service time versus NFS operation request rate for a constant NFS operation mix can be used as a means of comparison among various servers and server configurations.

Graphs which plot both NFS operation service time and NFS operation request rate take into account the fact that the actual service time associated with a high server NFS processing rate may not be as acceptable to a computing facility as the lower service time encountered at a lower server NFS operation processing rate. Additionally, NFS operation service time trends can serve as a rough, ballpark indicator of client application performance. Thus, graphs that plot both NFS operation service time and NFS operation request rate will assist a computing facility in answering the question "How will our application perform using this file server?" However, the answer will be approximate rather than exact.

The maximum server resource utilization levels produced by nhfsstone were not comparable with those obtained with the AWM approach. This can cause the average NFS request service time reported by nhfsstone to be optimistic (low). Optimistic response indications further obscure the relationship between the service times reported by nhfsstone and the AWM approach. This optimism is offset by the pessimistic nature of nhfsstone in defeating various client caches which tends to increase service times. The degree to which the pessimism balances the optimism could serve as an area for future work.

Nhfsstone complements, but does not replace, the AWM approach to NFS file server performance characterization given the non-linearity of NFS operation request rate identified in Section 3.3. In order for nhfsstone to exactly duplicate an actual workload, the non-linear degradation of a workload's NFS operation request rate must be known in advance and implicitly supplied to nhfsstone through its requested NFS operation rate input parameter.

The AWM approach continues to have merit in that, in addition to providing user-level client response information, the approach also tests large scale software and hardware interoperability. Further, the AWM approach does not require prior knowledge of a workload's NFS operation request rate degradation to associate client NFS request rate levels with given numbers of clients.

4.1 Possible Future Work

Several areas of future work are possible. As previously mentioned, the relationship between average NFS request service time as reported by nhfsstone and user-level task service time as reported by the AWM approach should be further investigated. The ability of nhfsstone to replicate other actual workloads should also be investigated.

Measurements of NFS operation distributions (mixes) associated with several computing environments should be made and distributed within the industry so that these NFS operation distributions can be consistently used to characterize an NFS server's performance in different computing environments. A checklist of NFS setup parameters (e.g., NFS timeout interval, etc.) should be defined to ensure that a consistent environment is established when servers are characterized with the various NFS operation distributions.

The nhfsstone utility might be further enhanced to more accurately reproduce actual workloads in terms of maximum server resource utilization levels. This would improve the accuracy of the average NFS request service time reported by nhfsstone so that more accurate predictions of client application response could be made. Rather than taking a single NFS mix and operation request rate as input, an enhanced version of nhfsstone might accept several mixes and request rates as input so that it could dynamically change the generated load during execution. Alternatively, an enhanced version of nhfsstone could generate a distribution of NFS operation request rates within limits set by the user. Both of these enhancements would assist nhfsstone in expanding its role into an actual workload abstraction/capture/replay tool targeted towards more accurate prediction of client application response than its current role as a load generator. These enhancements would allow nhfsstone to grow into a tool that would allow a computing facility to more accurately answer the question "How will our application perform using this server?"

5. ACKNOWLEDGMENTS

I am indebted to many people for their various contributions to this evaluation: Charlie Briggs and Chet Juszczak for their encouragement, insights, and comments; Paul Kotschenreuther for his comments and role as chief sounding-board; Santa Wiryaman for his comments and automation of the evaluation process; Judy Piantedosi for her assistance in test environment setup and WYSIWYG document preparation wizardry; Muhammad Ashraf and Doug St. Clair for their statistical prowess; Lucy Patton and Angela Richard for their technical writing critiques; all of those in LES-SECG Lab Services for their fast and accurate response to test equipment and configuration requests; and last but not least, Lee Allison, Paul Kruger, and Linda Chasey Vincent of LES-SECG management for their strong and constant support of this evaluation.

I would also like to acknowledge the understanding support of my wife, Laurie, our son Ted, our newborn son Joseph Brian, and Laurie's and my parents, without which this paper would not have been produced within deadline, as Joe was born in the midst of developing this paper.

6. REFERENCES

[BRIGGS88] Briggs, Charles, ''NFS Diskless Workstation Performance'', Digital Equipment Corporation Technical Report, February 24, 1988.

[KLEINROCK75] Kleinrock, Leonard, ''Queueing Systems Volume 1: Theory'', Wiley-Interscience, 1975.

[LEGATO89] nhfsstone NFS load generating program, Legato Systems Inc., Palo Alto, CA 94306.

[SANDBERG85] Sandberg, Russel, et al., ''Design and Implementation of the Sun Network Filesystem'', USENIX Summer Conference Proceedings, pp.119-130, June 1985.

[SHEIN89] Shein, Barry, et al., ''NFSSTONE - A Network File Server Performance Benchmark'', USENIX Summer '89 Conference Proceedings, pp. 269-274.

Bruce Keith is a Principal Software Engineer in the Low End Systems Systems Engineering Characterization Group at Digital Equipment Corporation. Since joining Digital in December 1986, he has been leading a project concerning the performance characterization of ULTRIX NFS file servers. Prior to joining Digital, Bruce developed systems software spanning UNIX, VMS, RSX-11M, and TOPS-10 operating environments for timesharing, academic, and technical OEM businesses. Bruce received his BS degree in Computer Science from Worcester Polytechnic Institute.

Reach Bruce at the Systems Engineering Characterization Group, Low End Systems; 129 Parker Street PKO3-1/D18; Maynard, Massachusetts 01754; or by e-mail at keith@oldtmr.enet.dec.com .

USENIX Summer Conference
June 11-15, 1990
Anaheim, California

John H. Hartman, John K. Ousterhout –
University of California at Berkeley

Performance Measurements of a Multiprocessor Sprite Kernel

ABSTRACT

This report presents performance measurements made of the Sprite operating system running on a multiprocessor. A variety of micro- and macro-benchmarks were run while varying the number of processors in the system, and both the elapsed time and the contention for kernel locks were recorded. A number of interesting conclusions are drawn from the results. First, the macro-benchmarks show acceptable performance on systems of up to five processors. Total system throughput increases almost linearly with the system size. Projections of the lock contention measurements show that the maximum performance will be reached with about seven processors in the system. Second, it is often difficult to predict the effect of a benchmark on particular kernel locks. It was anticipated that different benchmarks would saturate different kernel monitor locks. After running the benchmarks it was found that a single master lock was the biggest kernel bottleneck, and that one of the micro-benchmarks had saturated a different lock than the one at which it was targeted. The kernel locking structure has become so complex as the system has evolved that it is hard to determine cause and effect relationships. Third, although the kernel contains many locks, only a few of them are performance bottlenecks. Performance measurements such as those presented here allow the relevant parts of the kernel to be redesigned to eliminate the bottlenecks. Such a redesign is needed to allow the system to scale gracefully beyond about seven processors.

1. Introduction

Sprite is a network operating system being developed at Berkeley [6]. From its inception Sprite has been designed to run on a multiprocessor. To avoid performance bottlenecks due to kernel contention, the kernel is multi-threaded to allow more than one processor to execute kernel code at the same time. Exclusive access to kernel resources is ensured through the use of locks. There is a limit to the number of processors that can be in the kernel and doing useful work, however. Once a lock saturates, additional processors will not significantly increase system throughput. The goal of our study was to determine how well Sprite scales with the size of a multiprocessor, by running a variety of benchmarks and measuring both contention for kernel locks and overall system performance.

The benchmarks were chosen either to stress particular kernel locks or to resemble user workloads seen in the Sprite development environment. The former were used to measure how well the system behaves when locks became saturated, and to identify locks that are potential bottlenecks. This information can then be used to restructure the kernel to improve its behavior. The latter were used to measure the system's ability to scale while running realistic workloads, determining both the limit on system size and the effect of adding another processor to the system.

The results indicate that contention for the kernel context switch code is the biggest limiting factor to scaling the system. The lock protecting this code becomes very heavily utilized with only five processors in the system. Measurements of the realistic workloads indicate that the lock will become saturated with about seven processors in the system.

The rest of the paper is structured as follows. Section 2 describes the types of locks used in the Sprite kernel, and Section 3 describes the multiprocessor hardware used to obtain the measurements. The instrumentation added to the Sprite kernel is outlined in Section 4, and a description of the benchmarks is in Section 5. The results are in Section 6, followed by comments in Section 7 and concluding remarks in Section 8.

2. Kernel locks

In order for a multi-threaded kernel to function correctly it must contain mechanisms for providing both mutual exclusion and synchronization between

The work described here was supported in part by the Defense Advanced Research Projects Agency under contract N00039-85-R-0269 and in part by the National Science Foundation under grant ECS-8351961.

threads. Other efforts to design multiprocessor operating systems have used semaphores [2,4,8]. Semaphores are appealing because they can provide both mutual exclusion and synchronization, eliminating the need for separate mechanisms. Sprite, however, uses monitor-style locking and condition variables to provide these services.

There are two basic types of locks in the Sprite kernel: *monitor* locks and *master* locks. Monitor locks are used to implement monitors [9], with semantics similar to those in Mesa [5]. Monitor locks are acquired at the start of a procedure and released at the end. If a process tries to lock a monitor lock and another process has it locked already, then the process is put to sleep. The release of a monitor lock causes all processes that are waiting on the lock to be awakened and simultaneously try to reacquire the lock.

The other type of lock in the Sprite kernel is the *master* lock. Master locks are used in much the same manner as monitor locks, except that they are used to to provide mutual exclusion between processes and interrupt handlers. A master lock is simply a spin lock that is acquired with interrupts disabled. If a master lock is already in use when an attempt is made to grab it, then the processor retries the locking operation until it succeeds. Interrupts are disabled to prevent a situation where an interrupt is taken after a master lock has been acquired and the interrupt routine spins forever waiting for the lock to be released.

Locks have three different styles of usage in the Sprite kernel. These three styles correspond to different locking granularities. Fine-grained locks allow a high degree of concurrency, but they increase the number of locks that a particular thread must acquire, thereby decreasing its performance. Coarse-grained locks reduce the amount of concurrency, but improve the performance of a single thread. The trick in designing the kernel is deciding the placement and granularity of locks.

The coarsest granularity locks in the kernel are single locks that are used to protect sections of code. Locks used in this manner are referred to as *code locks*. If the lock is a monitor lock then the resulting construction is similar to the *monitored modules* described in [5]. An example of this style of use is the monitor lock around the Sprite virtual memory system. There is a single monitor lock that must be grabbed whenever a routine in the virtual memory system is called, thus synchronizing access to the virtual memory system as a whole.

The remaining two styles of usage are variations on a theme. They both associate locks with data rather than with sections of code. For this reason they are referred to as *data locks*. Data locks that use a monitor lock are referred to as *monitored objects* in the Mesa paper. One style of usage associates a lock with a data structure. In order to manipulate the data structure the lock must be held. A lock that protects a queue is an example. The Sprite file cache has a single lock that protects the various lists of cache blocks, such as the free list, dirty list, etc. In order to modify any of these lists the file cache lock must be held.

The finest granularity locks are associated with individual data objects. A processor must hold this lock before modifying the contents of the object. For example, each block in the Sprite file cache has a lock, and that lock must be held when accessing the contents of the block.

In addition to ensuring mutual exclusion between threads, Sprite must also provide a means for threads to wait for interesting conditions to occur. *Condition variables* are used for this purpose. If a process waits on a condition variable while holding a lock it will release the lock and be put to sleep. At a later time another process will signal the condition variable, causing all processes waiting on the variable to awake and try to reacquire the lock. This signaling mechanism has the same semantics as Mesa's *broadcast* facility.

Major Locks in the Sprite Kernel		
Name	Type	Description
sched_Mutex	Master	controls access to context switch code and run queue
handleTableLock	Monitor (code)	controls access to table of all known file handles
vmMonitorLock	Monitor (code)	lock around all virtual memory functions
perPCBLock	Monitor (data)	must be held when accessing contents of a process control block
pdevLock	Monitor (data)	controls access to individual pseudo-device handles
exitLock	Monitor (code)	provides exclusive access to code for destroying a process

Figure 1. Some of the major locks in the Sprite kernel. Monitor lock types are qualified by a designation in parentheses. *Code* indicates that there is a single lock protecting a critical section of code. *Data* designates those locks for which there is one lock per object. Handles are Sprite's equivalent to Unix's inodes. Pseudo-devices are explained later in this paper.

Figure 1 is a list of the major locks in the Sprite kernel. There are many other locks in the kernel, such as locks associated with the file system cache, system timers, and the RPC system, but none of these have a significant impact on system performance.

3. The SPUR Hardware

This section outlines some of the features of the hardware used in the study. Sprite does not require any special hardware support for running on a shared-memory multiprocessor, other than an atomic test-and-set operation and coherent processor caches. Details of the hardware are only provided to allow comparisons to be made to more familiar machines.

At the time of this study the only multiprocessor that Sprite supported was the SPUR multiprocessor [3]. SPUR is a RISC microprocessor developed at Berkeley as part of a project to study the impact of adding symbolic processing support and multiprocessing to RISC architectures. Individual SPUR processor boards can be combined to form a shared-memory multiprocessor. The machine used in this study has 32 Mb of shared memory and up to five processors. Each SPUR processor board has a 128-kbyte data cache that is kept consistent by hardware.

The SPUR prototype used in this study is not a particularly fast machine. The processor cycle time is 140 ns. Due to a design error the on-chip instruction buffer is not functional, causing instruction fetches to require several cycles. As a result, a SPUR processor can execute about 1.5 native MIPS. Furthermore, a lack of compiler optimization leads to inefficient code. We estimate that the resulting performance is equivalent to about 0.5 VAX MIPS.

4. Instrumentation

Contention for kernel resources was measured by collecting information on lock behavior. We added fields to each lock to record the number of successful lock acquisitions (referred to as *hits*), and the number of unsuccessful attempts to acquire the lock (*misses*). For some types of locks, such as the locks associated with process control blocks, the statistics for an individual lock aren't as useful as those for the type as a whole. For this reason the individual lock counts were consolidated and recorded on a per-type basis. Recording the information on a per-type basis also makes it easy to handle locks that are created and destroyed dynamically. The lock counts for these transient locks are added to the total for the type when the lock is destroyed.

The definition of what constitutes a lock miss is different for monitor locks and master locks. For master locks there is at most one miss per hit. If a process misses on a lock it spins until the lock is free, then locks it. This sequence is counted as one miss, followed by a hit. On the other hand, it is possible for monitor locks to have more misses than hits. If multiple processes are waiting when a lock is released then they will all be awakened and will attempt to grab the lock. Only one will succeed and the rest will go back to sleep. If a process is awakened in this fashion and does not get the lock a miss is recorded. This makes it possible for the number of misses on a monitor lock to be greater than the number of hits.

In addition to instrumenting the locks we also created new system calls to clear the lock information and to copy the information to user-level. These two calls were used to reset the lock information at the beginning of a test and gather the information when it completed.

5. Benchmarks

All of the benchmarks are intended to stress the operating system. Compute-bound applications were avoided for that reason. The benchmarks can be divided into two classes. Realistic workloads are represented by the *macro-benchmarks*. These benchmarks are comprised of real programs that represent the Sprite development environment. The behavior of the kernel while running the macro-benchmarks is indicative of its behavior under real workloads, and allows projections to be made of maximum system size.

Individual parts of the kernel were stressed by running a series of *micro-benchmarks*. For example, a micro-benchmark may consist of repeated forking of children, or repeated message passing between processes. Such repetitive behavior is rarely seen in real programs, but it does allow the behavior of different parts of the kernel to be isolated and measured.

Macro-benchmarks

PmakeInd

The *pmakeInd* benchmark is intended to be representative of a multi-user environment with multiple independent compilations taking place. *PmakeInd* recompiles Csh from its sources using the *Pmake* program. Pmake is similar to the UNIX "make" utility, except that it runs the compilations in parallel whenever possible [1]. The *pmakeInd* benchmark runs a separate instance of Pmake on each processor and each instance uses separate copies of the sources. This eliminates the Pmake program and contention for the source files as causes of performance degradation. Each instance of Pmake runs two compilations concurrently, ensuring that each processor remains highly utilized.

Pmake

The *pmake* benchmark is also a compilation of the Csh sources, except that only one instance of Pmake is run, rather than one per processor. Pmake attempts to use all of the processors in the system by running two compilations per processor. The purpose in running this benchmark is to see how well a single parallel application can harness the processing power available in a multiprocessor. Ideally the application will realize linear speedup as the number of processors is increased. A speedup that is less than linear is due to either contention in the kernel or lack of concurrency in the application. Since we are primarily interested in the former effect rather than the latter, the final link of the Csh binary was eliminated from the benchmark. This increased the concurrency inherent in the computation and increased contention for kernel resources.

Troff

The *Troff* benchmark consists of running the text formatting program Troff on the man page for Csh. The resulting output is sent to /dev/null. Each processor runs a different instance of Troff.

Micro-benchmarks

Twelve micro-benchmarks were run, of which only six are discussed in this paper. The other six pertained to stressing the file system, by reading files, opening files, etc. The results obtained were not significantly different from the first six and are therefore omitted in the interest of brevity.

Fork

The *fork* benchmark consists of a process that repeatedly forks off a child process using the *fork()* system call. The parent waits for the child to die before forking another child. The child process exits immediately. This benchmark should stress the process creation and destruction components, and the kernel virtual memory system. The context-switch code will also be heavily utilized.

Fexec

The *fexec* benchmark is similar to the *fork* benchmark. A parent process repeatedly forks off children, waiting for each one to die before forking the next. The child process uses the *exec()* routine to create another instance of itself instead of exiting. This new instance then exits, allowing the parent to continue. The *fexec* benchmark should stress the same components as the *fork* benchmarks. Any differences in behavior are due to the call to *exec()*.

Mem

Mem is a benchmark that stresses the virtual memory system. The process repeatedly increases the size of its heap using the *sbrk()* system call. The new heap pages are not touched. Once the process reaches a maximum size it execs itself and starts over. *Mem* is targeted at the virtual memory system, particularly the component that is responsible for maintaining process page tables.

Cswitch

The *cswitch* benchmark consists of two processes that pass a byte back and forth using a pair of pipes. If this pair of processes is run on a single processor, then two context switches are required per round trip and the context switch code should receive heavy utilization. The context switches can be avoided on a multiprocessor if the processes are run on different processors. In order to ensure that the context switches occur one pair of processes is run per processor in the system.

Pdevtest

Pdevtest stresses the pseudo-device implementation. Pseudo-devices are like devices except that they are implemented by user-level server processes [7]. They provide a mechanism for allowing trusted services, such as display servers and network communication protocols, to be implemented at user-level rather than in the kernel. A client process accesses a pseudo-device in the same manner as a real device. The kernel then forwards the request to the user-level server, instead of to a kernel device driver. Pseudo-devices are similar to the UNIX "pty" mechanism, the only difference being that all operations on a pseudo-device, such as open, close, and ioctls, are passed through to the server process.

The *pdevtest* benchmark creates one pseudo-device that is written to by multiple clients. Each client repeatedly writes one byte to the server.

PdevtestInd

This variation on the *pdevtest* benchmark creates multiple pseudo-devices. One server and one client is created for each instance of the benchmark that is run (i.e for each processor in the system). The intent is to see if there are any performance bottlenecks in the pseudo-device implementation that occur when there is a single server vs. multiple servers.

6. Results

The results are divided up into sections on elapsed time for running the benchmarks, calculations of the incremental throughput for each additional processor, monitor lock behavior, and master lock behavior. These measurements indicate that the micro-benchmarks suffer heavily from saturation of critical resources. The macro-benchmarks also experience performance degradation, but to a lesser degree.

Elapsed time

The elapsed time ratio for the macro-benchmarks is shown in Figure 2. The ratio is calculated by dividing the time it takes a system with *n* processors to finish *n* benchmarks by the time it takes a uniprocessor to finish one benchmark. For all but the *pmake* benchmark the workload is scaled with the number of processors, so an ideal system would have a constant ratio of 1.0. The curve for the *pmake* benchmark looks different because the workload is not scaled with the number of processors. The compilation of the Csh sources always runs faster with more processors. The ideal elapsed time ratio for the *pmake* benchmark is 1/*n*, where *n* is the number of processors in the system. The measured ratio for the *pmake* benchmark is very close to the ideal, falling behind only when there are five processors in the system.

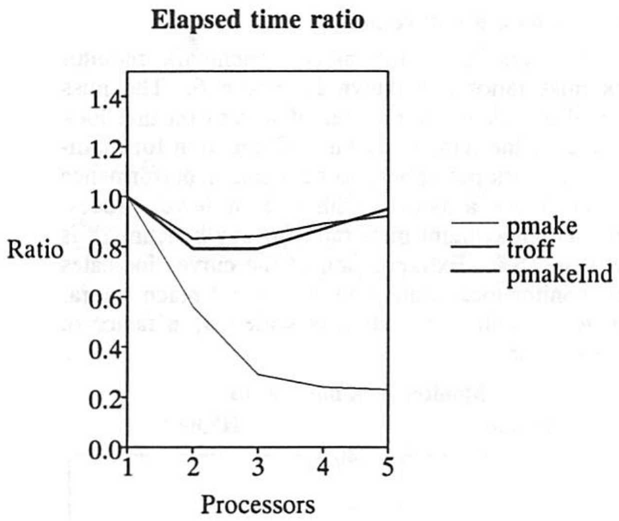


Figure 2. A graph of the ratio of the multiprocessor elapsed time to run the macro-benchmarks to the uniprocessor elapsed time. The workload was scaled with the system size (except for the *pmake* benchmark), causing the ratio for an ideal system to be a horizontal line at 1.0. A ratio that is greater than 1.0 indicates that the increase in system throughput is less than linear. The *pmake* benchmark has a fixed workload, hence its ideal curve should be *1/n*, where *n* is the number of processors.

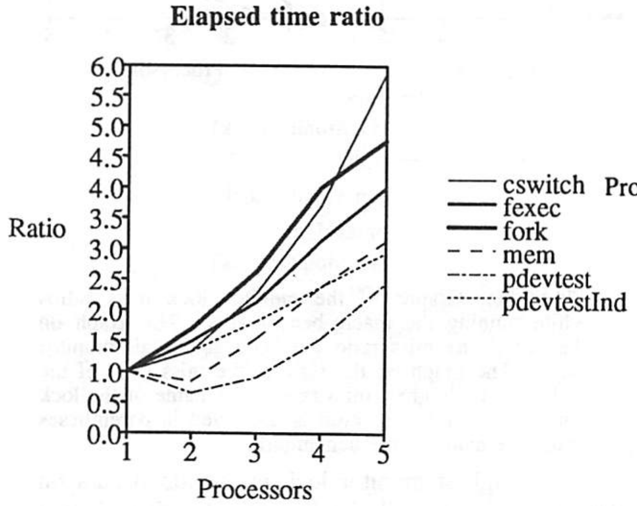


Figure 3. This is a plot of the ratio of the multiprocessor elapsed time to run the micro-benchmarks to the uniprocessor elapsed time. As in Figure 2 an ideal system would have a ratio of 1.0. All of the benchmarks exhibit an increase in the elapsed time as the system size increases.

The *troff* and *pmakeInd* benchmarks have elapsed time ratios that are actually better (i.e. less) than the ideal value of 1.0. This is because there is a certain amount of background processing that must be done that is independent of system size. This extra work is due to other processes running on the system and interrupts. As the number of processors in the system is increased, the per-processor load induced by background processing is decreased, allowing each processor to allocate more cycles to the benchmark. While running the *troff* and *pmakeInd* benchmarks the slowdown from kernel contention was more than offset by the amortization of the background processing.

Figure 3 shows the elapsed time ratio for the micro-benchmarks. Contention causes the elapsed time of all the benchmarks to increase over the range of system sizes tested. For two of the benchmarks the elapsed time initially decreases due to amortization of background processing costs, but this benefit is eventually outweighed by contention for kernel resources.

The *cswitch* benchmark has a worse than linear slowdown: a system with five processors actually takes longer to complete the work than a uniprocessor should. None of the other benchmarks do this badly, but neither do they come close to the ideal ratio. All of the benchmarks show a steady increase in the elapsed time ratio once there are three processors in the system. This suggests that the micro-benchmarks are severely affected by contention for kernel resources and the system throughput does not increase significantly if the system is scaled beyond three processors.

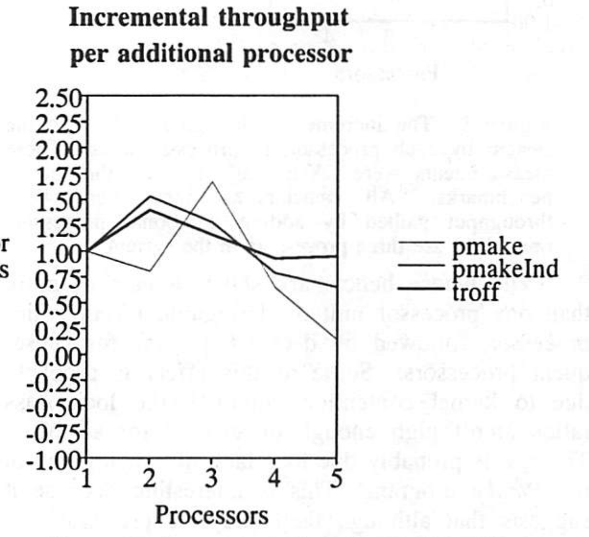


Figure 4. Incremental throughput per additional processor while running the macro-benchmarks. Two of the benchmarks have almost constant incremental throughput, indicating that the total system throughput is proportional to the size of the system.

Incremental throughput per additional processor

The incremental throughput of an additional processor is the net amount of work per unit time that the processor adds to the system. Throughput is measured in *processor units*, which is the amount of work per unit time done by a uniprocessor. Incremental throughput is derived by taking the throughput with *n* processors in the system, and subtracting the throughput with *n* - 1 processors. The result is then normalized to the throughput of a uniprocessor to obtain the incremental throughput in processor units.

Figure 4 shows the incremental throughput per processor for the macro-benchmarks. The independent compilation of Csh (*pmakeInd*) and the troff of the Csh man page (*troff*) curves both display a gain of more than one processor unit when the second processor is added. As mentioned previously, this is due to amortization of background processing. The *troff* benchmark does the best of all, maintaining an incremental gain of almost one processor unit even for the fifth processor.

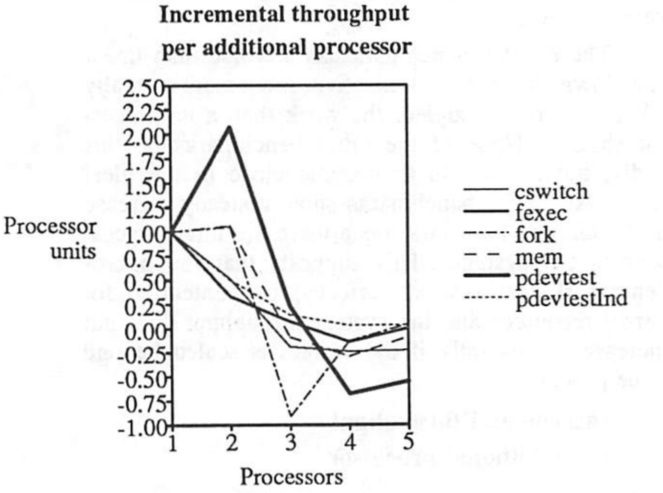


Figure 5. The incremental throughput added to the system by each processor, in processor units. These measurements were taken while running the micro-benchmarks. All benchmarks show very little throughput gained by adding additional processors once there are three processors in the system.

The *pmake* benchmark shows a gain of more than one processor unit of throughput for the third processor, followed by decreasing gains for subsequent processors. Some of this effect is probably due to kernel contention, although the lock miss ratios aren't high enough to account for all of it. The rest is probably due to a lack of concurrency in the *Pmake* program. This is interesting because it suggests that although there are five processors in the system, *Pmake* cannot make effective use of them.

The micro-benchmarks all suffer from severe degradation in elapsed time as the number of processors is increased, therefore we would expect the incremental throughput per processor to diminish as well. This result is seen in Figure 5. The throughput gained by adding additional processors drops off rapidly so that the third processor does not add much, if any, processing power to the system. For a few of the benchmarks the third processor actually produces negative processor units of throughput, in effect slowing the system down. At this point the scheduler lock has become saturated, preventing additional processors from doing anything useful.

Monitor lock measurements

The graphs of the macro-benchmark monitor lock miss ratios are shown in Figure 6. The miss ratio of a lock is the number of misses on that lock divided by the number of hits. Contention for monitor locks does not appear to be a major performance bottleneck for a system with five or fewer processors. The maximum miss ratio for any benchmark is less than 18%. Extrapolation of the curves indicates that monitor lock contention should not reach saturation levels until the system is scaled by a factor of three or four.

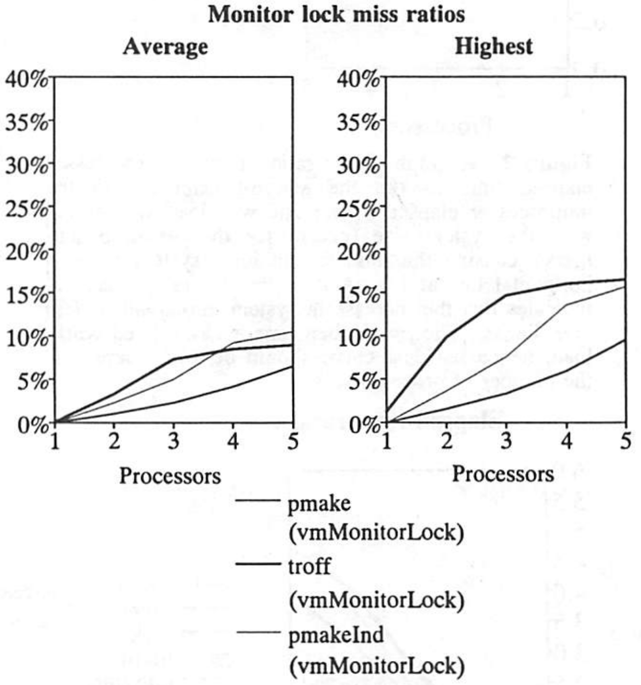


Figure 6. Graphs of the monitor lock miss ratios while running the macro-benchmarks. The graph on the left is the miss ratio averaged across all monitor locks. The graph on the right is the miss ratio of the lock with the highest miss ratio. The name of the lock with the highest miss ratio is displayed in parentheses under the name of the benchmark.

The highest monitor lock miss ratio occurs on the same lock for all the benchmarks. The kernel has a single monitor lock, vmMonitorLock, that surrounds the entire virtual memory system. Any routines that modify the virtual memory state must hold this monitor lock. When the virtual memory system was written the emphasis was on correctness, rather than concurrency, hence vmMonitorLock's monolithic nature. However, the benchmarks suggest that system performance on large multiprocessors could be improved by replacing the single vmMonitorLock with several locks on individual data structures.

The monitor lock miss ratios while running the micro-benchmarks are shown in Figure 7. Most of the curves have small slopes and small maximum values. Exceptions are the *pdevtest* and *cswitch* benchmarks. The causes of the contention can be found by examining the kernel code. The *pdevtest* benchmark spends a lot of its time copying the data from the client's address space to the server's. A monitor lock associated with the server process (perPCBLock) is held during the copy. If there are multiple clients then this lock is a critical resource. With more than three clients the lock is held all of the time, causing the miss ratio to reach almost two hundred percent. Each time a process releases the lock more than one process is awakened causing more total misses than hits.

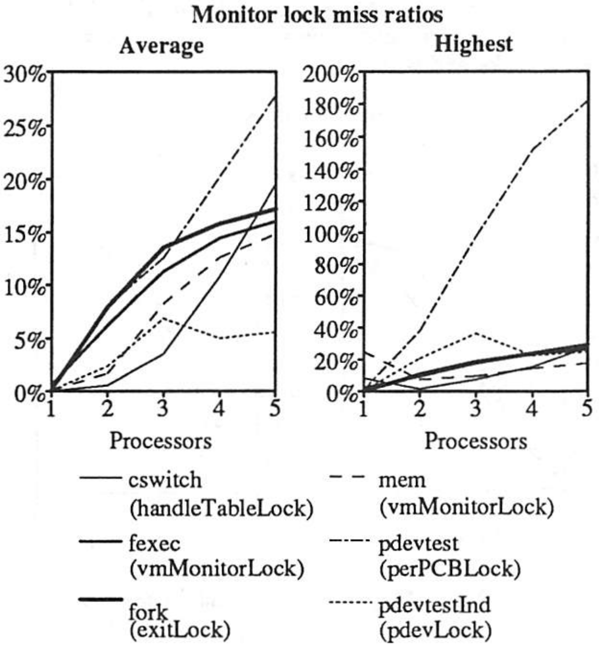


Figure 7. The average and highest monitor lock miss ratios while running the micro-benchmarks. Note that the vertical scales on the two graphs are not the same. Benchmarks whose curves have a steep slope will not scale well to larger systems due to saturation of a monitor lock. The initially high values for some benchmarks on a one processor system are due to all of the benchmark processes exiting at once, causing a high miss ratio although the absolute number of hits and misses is very low.

The *cswitch* benchmark uses pipes to pass a byte between processes. Each time a pipe is accessed a monitor lock around the file handle table (handleTableLock) in the kernel is grabbed. Although different pairs of processes use different pipes, they all need to grab the handle table monitor lock, causing a bottleneck. This is a surprising result, since the *cswitch* benchmark was intended to stress the context switch code. It is not always easy to predict what effect a particular kernel lock will have on a benchmark's performance.

Two other locks show up in the graph of the highest monitor lock miss ratio. ExitLock and pdevLock have the highest miss ratio of any monitor lock while running the *fork* and *pdevtestInd* benchmarks, respectively. The miss ratios are not high enough, however, to warrant replacing each of these locks with multiple locks unless there are many more processors in the system.

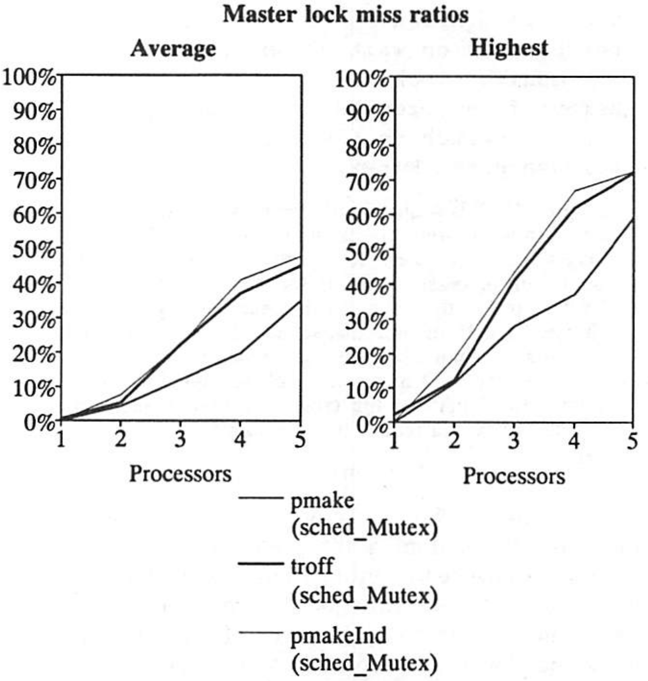


Figure 8. Graphs of the master lock miss ratios while running the macro-benchmarks. All of the curves have a steep slope due to contention for the scheduler lock.

Master lock measurements

The graphs in Figure 8 are plots of the master lock miss ratios for the macro-benchmarks. All of the curves show increasing amounts of contention as the size of the system is increased. All of this contention is due to sched_Mutex, the master lock around the scheduler. Sched_Mutex has a higher miss ratio than any other lock in the kernel for almost all of the benchmarks. This is surprising, since our first intuition was that various monitor locks would saturate first. It turns out that sched_Mutex is used in many different places in the kernel. Its main function is to provide mutually exclusive access to the run queue, but it is also used for other purposes, including synchronization when a miss occurs on a monitor lock (see Figure 9). As a result, an increase in the miss rate for monitor locks will increase the contention for sched_Mutex.

The graphs in Figure 10 are plots of the master lock miss ratio for the micro-benchmarks. The slopes of the curves are quite steep – all benchmarks have an average miss ratio on a five processor system that is between fifty and eighty-five percent. The highest miss ratio for any lock was greater than eighty percent for all benchmarks, and in some cases was higher than ninety percent. Once again this contention is for sched_Mutex. Although the micro-benchmarks were targeted at an array of kernel monitor and master locks, they all piled up on the scheduler lock. Clearly the importance of this single lock must be reduced.

```
Miss on monitor lock.
Lock sched_Mutex.
Put process on wait queue for
    monitor lock.
Remove first process from ready queue.
Context switch to the ready process.
Release sched_Mutex.
```

Figure 9. This sequence of events occurs when a process misses on a monitor lock. Note that sched_Mutex is used to synchronize access to the queue of processes waiting for the monitor lock, as well as the ready queue. This unnecessarily increases the total length of time that sched_Mutex is held. The addition of individual master locks to synchronize access to the wait queue for each monitor lock would reduce the length of the critical section protected by sched_Mutex and reduce its utilization.

7. Comments

A running Sprite kernel contains anywhere from five hundred to a thousand locks. Of these locks, two repeatedly suffered more contention than the others. These two locks, sched_Mutex and vmMonitorLock, account for most of the contention for kernel locks and represent serious obstacles to better system performance. Both of these locks are code locks that protect large regions of the kernel code. In order to reduce their impact they must be replaced by several data locks with a finer granularity. It is not necessary to have single locks around the virtual memory system and the context switch mechanism. Replacing these locks with data locks on the various kernel structures will increase the concurrency and system performance.

There are a number of factors that cause system performance to depend heavily on contention for a few locks. The first is ease of design. It is much simpler to design a system with a few locks than it is to design one with many locks. Multiple locks may increase the concurrency, but they also increase the chance of introducing race conditions and deadlocks. A single lock should be replaced by multiple locks only if performance measurements indicate that the single lock is a bottleneck.

The second factor that causes performance-critical locks is that they develop during the evolution of the kernel. Race conditions and deadlocks tend to occur as new features are added to the kernel. When a kernel developer is faced with such an unwanted side-effect they typically rewrite their new feature to hold the most prominent lock they can find. In this manner prominent locks tend to become more important, until they are held in many places throughout the kernel for many different reasons. Such is the case with sched_Mutex. Its influence grew as the kernel was modified and synchronization problems arose.

The underlying cause of both the design and development problems is the lack of tools. The graph of lock dependencies in the Sprite kernel is fairly complex. Tools are needed to help understand the synchronization requirements of the various kernel resources and where to place locks to satisfy those requirements. Once the locks are in place, tools are needed to measure the performance of the locks in order to find the performance bottlenecks. Our measurements of kernel locks in Sprite found that it is not always obvious which locks will be bottlenecks and why.

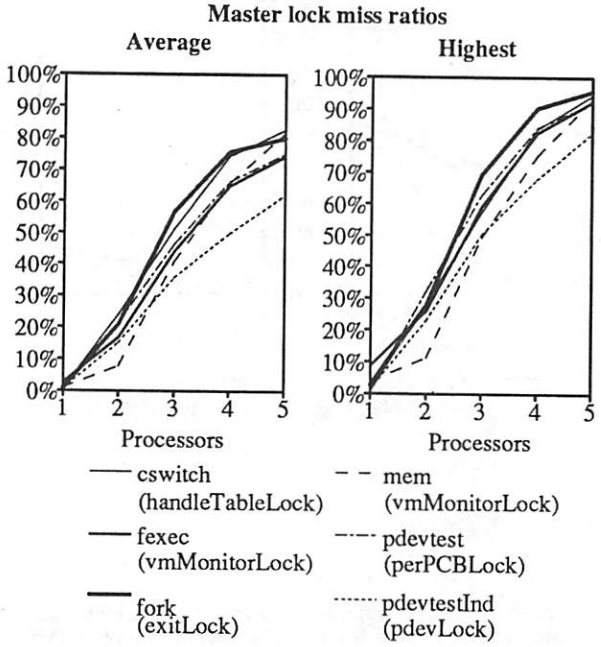


Figure 10. Graphs of the master lock miss ratio while running the micro-benchmarks. The steep slopes of the curves are due to a heavy contention for the master lock around the scheduler.

8. Conclusion

The elapsed time and incremental throughput measurements for the macro-benchmarks indicate that the Sprite kernel gives acceptable performance on a machine with up to five processors. All of these benchmarks showed an almost linear speedup as the system was scaled, and two of them showed almost constant incremental throughput per processor. The performance degradation of the *pmake* benchmark was primarily due to sequential processing in the application, rather than kernel contention. This underscores the difficulty of writing a single application that can make full use of a multiprocessor's processing power.

Although the macro-benchmarks exhibited suitable performance increases for the system sizes that were measured, the sched_Mutex master lock approached saturation. With five processors in the system its miss ratio was close to seventy percent, indicating that the lock suffered very heavy contention. From this it would appear that without elimination of some of the more important bottlenecks the kernel will not support more than seven processors in the system efficiently, except for compute-bound applications.

Prior to undertaking this study we had assumed that the kernel monitor locks were the biggest performance bottlenecks. In particular, it was feared that the single monitor lock around the virtual memory system posed the biggest obstacle to scaling the number of processors in the system. It came a surprise that a master lock, sched_Mutex, was more heavily utilized than the virtual memory system lock. We were also surprised when micro-benchmarks stressed unintended locks. The behavior of kernel locks is difficult to predict, even to the people who designed the system. Clearly there is a need for tools to help system developers to better understand and modify the locking structures of multiprocessor operating systems.

9. Acknowledgements

We would like to thank Ken Lutz for putting together and maintaining the SPUR prototype, and Mendel Rosenblum for his efforts in getting both Sprite and SPUR to run reliably.

10. References

1. A. Boor, PMake – A Tutorial, Unpublished, June 1, 1988.
2. S. J. Buroff, Multiprocessor UNIX Operating Systems, *ATT Bell Laboratories Technical Journal 63*, 8 (October 1984), 1733-1749.
3. M. D. Hill, S. J. Eggers, J. R. Larus, G. S. Taylor, G. Adams, B. K. Bose, G. A. Gibson, P. M. Hansen, J. Keller, S. I. Kong, C. G. Lee, D. Lee, J. M. Pendleton, S. A. Ritchie, D. A. Wood, B. G. Zorn, P. N. Hilfinger, D. Hodges, R. H. Katz, J. Ousterhout and D. A. Patterson, SPUR: A VLSI Multiprocessor Workstation, Computer Science Division Technical Report UCB/Computer Science Dpt. 86/273, December 1985.
4. Hierarchical Ordering of Sequential Processes., in *Operating Systems Techniques*, 1972, 72-93.
5. B. W. Lampson and D. D. Redell, Experiences with Processes and Monitors in Mesa., Vol. 23, February, 1980,.
6. J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson and B. Welch, The Sprite Network Operating System, *IEEE Computer 21*, 2 (Feb. 1988), 23-36.
7. B. B. Welch and J. K. Ousterhout, Pseudo-Devices: User-Level Extensions to the Sprite File System, *Proc. of the 1988 Summer USENIX Conf.*, June 1988, 184-189.
8. A Strategy for SMP Ultrix, *Usenix Conference Proceedings*, June 1988.
9. Monitors: An Operating System Structuring Concept, *Communications of the ACM 17*, 10 (October 1974,), 549-557.

John H. Hartman is a Ph.D candidate in the Department of Electrical Engineering and Computer Sciences at the University of California, Berkeley. He is currently a member of the Sprite network operating system project. His interests include operating systems, high-performance networks, and computer architecture. He received an Sc.B. in computer science from Brown University in 1987.

John K. Ousterhout is a Professor in the Department of Electrical Engineering and Computer Sciences at the University of California, Berkeley. His interests include operating systems, distributed systems, user interfaces, and computer-aided design. He is currently leading the development of Sprite, a network operating system for high-performance workstations. In the past, he and his students developed several widely-used programs for computer-aided design, including Magic, Caesar, and Crystal. Ousterhout is a recipient of the ACM Grace Murray Hopper Award, the National Science Foundation Presidential Young Investigator Award, the National Academy of Sciences Award for Initiatives in Research, the IEEE Browder J. Thompson Award, and the UCB Distinguished Teaching Award. He received a B.S. degree in Physics from Yale University in 1975 and a Ph.D. in Computer Science from Carnegie Mellon University in 1980.

Reach both authors at University of California at Berkeley; Computer Science Division; Electrical Engineering and Computer Sciences; University of California; Berkeley, CA 94720.

USENIX Summer Conference
June 11-15, 1990
Anaheim, California

Montage: Breaking Windows into Small Pieces

Paul Haahr – Princeton University

ABSTRACT

Window systems are hard to program because they involve connecting an asynchronous world, where a mouse may move or a key may be pressed at any moment, to programs which execute synchronously. This requirement has led to the use of the ''inverted program structure'' style of programming, which adds complexity to the underlying code. The Montage window system eliminates this complexity by providing a programming model that uses sequential fragments of code connected by synchronous I/O. The system and its clients are written in an extension to the Scheme programming language which supports concurrency.

In the same way as useful routines can be built up from simple programs by composing Unix pipelines, sophisticated applications can be created in Montage by connecting simple, ''lightweight'' processes. In addition, programs may be trivially (and transparently) modified by adding processes that filter input to or output from applications. This use of concurrency and composability is isomorphic to ''object-oriented'' programmingthings that would be objects in other window systems, such as window decoration (borders, title bars, etc.) or menus are processesbut, in practice, Montage processes are easier to write than objects in other window systems.

1. Introduction

Most windows systems are large programs with complicated programmer interfaces. Montage is window system that was designed to reduce the complexity of both application programs and the window system itself.

Traditional window systems are hard to write programs for largely because they force programmers to program in sequential languages which interact with an inherently asynchronous world where mouse movements or key clicks can happen at any time. The conventional approaches to managing this concurrency are structuring programs as event loops, or using the so-called ''inverted program structure,'' where the event loop is in a library rather than in a user's code. Rather than trying to fit the input to a window system into a sequential model, Montage whole-heartedly accepts the asynchrony. The Montage model derives from Hoare's Communicating Sequential Processes (CSP),[6] by way of Cardelli and Pike's Squeak[1] and Pike's Newsqueak.[12]

Montage's graphics model is also designed to make applications easier to write. Windows, which may be opaque or translucent, provide retained storage so an application programmer never has to respond to a redraw request from the server. The graphical interface is a small set of fundamental operations based on Porter and Duff's compositing algebra.[13] The current implementation of the graphics library only supports monochrome displays, but the model generalizes to color systems.

Montage is organized, like most current window systems, along the lines of a client-server model. The window system itself, the ''server,'' runs as one UNIX process, and communicates by either pipes or TCP/IP connections with application programs, the ''clients.'' Similar to NeWS[7] and the Blit,[9] but distinct from the X Window System,[17] application programs are split in two parts: one piece which runs within the address space of the server, and another which runs in a separate address space, possibility on a remote system.

The server is an interactive Scheme compiler, linked with about 2500 lines of C code to manage the interface with UNIX system calls and handle the low-level graphics operations. The rest of the server plus a basic window manager and several user interface objects (e.g., menus, window decoration) consists of about 1000 lines of Scheme.

The Montage server currently runs on Motorola 68020 based Sun workstations. Except for the low-level graphics library, which is written for the 68020, the code has run without difficulty on other machines. The client code is portable and runs on a variety of UNIX systems.

Section 2 of this paper covers the concurrent programming model. The graphics library is discussed in Section 3. Section 4 covers the construction of the window system itself.

2. Concurrent Programming in Scheme

Scheme is a lexically scoped, properly tail recursive dialect of Lisp. Scheme was chosen as the base language for Montage because:

- Scheme supports closures and higher-order functions.
- As an interactive language, the semantics for dynamically loading code into a running program are clear.
- It is possible to write the concurrency routines within Scheme itself.
- Scheme is garbage collected, which is necessary for a window system that runs (potentially bug-infested) application code inside the server.
- Dynamic type checking was seen as beneficial for prototyping pieces of the system when interfaces were changing.
- High quality implementations exist.

Scheme is defined in reference 14; reference 2 provides a good introduction to the language.

Montage Processes

The concurrent programming model in Montage is a method for structuring the system and does not imply anything about the implementation; the concurrency is virtual in the same way as timesharing on a uniprocessor machine is, and no true parallel execution necessarily occurs.†

Programs are structured as collections of (so-called "lightweight") processes running within the server that communicate with each other over untyped channels. Processes themselves are anonymous: there is no way to name a process, query its state, or make it abort except by sending or receiving messages over channels. Processes are started with the `(spawn)` macro, which is called with a block of code to execute in a new process. Spawn returns immediately.

Scheme has no built-in support for concurrency, but the language defines a construct named `call-with-current-continuation`, or `call/cc`, which makes it possible to implement coroutines, exception mechanisms, or other unusual control flow mechanisms, in a portable way.

The form of a `call/cc` operation is

(`call/cc` *procedure-a*)

where *procedure-a* must be a procedure that takes one argument, say, *procedure-b* which is also a procedure of one argument. If the call to *procedure-a* returns without calling *procedure-b*, `call/cc` uses the value returned by *procedure-a* as its return value. If, on the other hand, *procedure-a* does call *procedure-b*, *procedure-a* stops executing, the value passed to *procedure-b* is used as the return value of the `call/cc`, and execution continues from the point where `call/cc` was called. This mechanism is used in Montage to implement context switching and the process abstraction.

It would not be too much a stretch for a C programmer to consider `call/cc` a variant on `setjmp` with the procedure passed to `call/cc`'s argument corresponding to a matching `longjmp` with the appropriate jump buffer. The fundamental difference between the two is that C's `longjmp` is restricted (in the absence of assembly language trickery) to working within a stack disciplinea function may only jump to an ancestor in the call historywhereas Scheme imposes no such restriction.

Communication Primitives

A channel is created by calling the function `(make-chan)`. Channels are to Montage processes what pipes are to UNIX processes, the glue which connects processes. (Unlike UNIX processes there are no implicit channels analogous to standard input, output, and error.) The notation for receiving from a channel is

```
(<- channel)
```

and sending is

```
(-> channel value).
```

Channels are synchronous rendezvous points: a process that sends on a channel blocks until another process reads from the channel. If two processes are both blocked waiting to receive (or send) on a channel, the one that actually gets woken when some other process sends (receives) is picked non-deterministically. There is no way to determine if a send or receive will block, or, in fact, checking afterwards it did block, without using some out of band mechanism like the system clock.

Multiplexing channels is done with `select`. Select derives from the non-deterministic guard of CSP and the select statement of Newsqueak. Unlike CSP, where boolean tests can precede only receive operations and Newsqueak, where any tests must lie outside the select statement, a clause of a select may be guarded by an arbitrary number of conditionals in addition to specifying a potential communication. Select takes as its argument a series of *clauses*, where *clauses* are defined by the grammar

†Currently, processes are non-preemptively scheduled. This is a bug, not a feature.

```
select
  : (select clauses)
clauses
  :
  | (guard expression ...) clauses
guard
  : (if test) guard
  | (-> channel value)
  | value = (<- channel)
  | channel = (->* list-of-channels value)
  | value channel = (<-* list-of-channels)
```

When a select is executed, the if parts (where present) of all clauses are evaluated; send or receive operations on channels mentioned in clauses where the if part is true or not present are initiated. If no communications can complete, the select blocks until one can. If or when a communication completes, the expressions associated with that guard execute. If multiple channels can complete at any time, one is picked at random. The `<-*` and `->*` forms are used to select on any one of the channels on a list. The value of the entire select is the value of the last expression evaluated. Inside the body of code following each guard, the symbols to the left of the = are bound to the value and/or channel used in the communication.

Standard UNIX file operations, such as open, close, read, and write, are available from Scheme code. UNIX I/O calls are deferred until all running Scheme processes block, when a call to UNIX select is made.

One interesting side effect of running processes in a garbage collected environment is that if a set of processes deadlockall are waiting for communications to complete, and none of the channels that are reachable outside the set of processesthey are garbage collected. This can be very convenient, because processes do not have to be told explicitly to quit when the rest of the system is done with them; instead, the processes that communicated with them just ignore the channels connected to the processes in question and the resources used by those processes are reclaimed.

On the other hand, it is disconcerting to debug a system where processes that deadlock just disappear: a running program can stop and go back to the evaluator prompt without leaving a trace behind if it enters a deadlocked state. Deadlock prevention, like ensuring the absence of all other forms of bugs, is up to the programmer.

Examples

A simple example of the use of process, adapted from Newsqueak,[12] is a sieve of Eratosthenes prime number generator.

```
(define (counter chan n)
  (-> chan n)
  (counter chan (1+ n)))
```

```
(define (filter-multiples n in out)
  (let ([m (<- in)])
    (if (not (zero? (remainder m n)))
      (-> out m)))
  (filter-multiples n in out))
(define (sieve in out)
  (let ([p (<- in)] [c (make-chan)])
    (-> out p)
    (spawn (filter-multiples p in c))
    (sieve c out)))
(define (make-sieve)
  (let ([c (make-chan)]
    [prime (make-chan)])
    (spawn (counter c 2))
    (spawn (sieve c prime))
    prime))
```

Running the sieve gives: (the evaluator's responses are shown in italics)

```
> (define primes (make-sieve))
> (<- primes)
2
> (<- primes)
3
> (<- primes)
5
> (<- primes)
7
> (<- primes)
11
```

A buffer process for a channel written using select is:

```
(define (buffer size in)
  (let ([out (make-chan)]
        [buf (make-vector size)])
    (define (incr n)
      (modulo (1+ n) size))
    (define (do-buffer count lo hi)
      (select
        [(if (< count size))
         v = (<- in)
          (vector-set! buf hi v)
          (do-buffer (1+ count)
          lo (incr hi))]
        [(if (> count 0))
         (-> out (vector-ref buf lo))
         (vector-set! buf lo '())
         (do-buffer (1- count)
         (incr lo) hi)]))
    (spawn (do-buffer 0 0 0))
    out))
```

3. Graphics Model

Graphics in Montage are based on the notion of compositing images. Every image has two parts, one for the data, which contains the color of each pixel, and one for a matte, which contains the degree of opacity for each pixel. In general, one uses a matte channel with as many bits per pixel as are provided in display hardware per channel of data: for example, a monochrome display will have one bit per pixel of matte as well as data, while a 24-bit RGB display would typically have 8-bits for the matte. Since Montage currently runs on only monochrome hardware, the data and matte portions of each image can be thought of as simple, 1-bit deep bitmaps. In the 1-bit model, if a matte bit is off, the image is clear for that pixel; if it is on, the pixel is opaque and has the color specified in the data bitmap.

Each image has both a data and matte section. Since storing two window-sized bitmaps for every layer is wasteful, especially in the presumably common case of fully opaque windows, Montage introduces an idea called "cards." A card is an infinite plane extending in all dimensions of one value, that may be used in lieu of the bitmaps for either (or both) of the data or matte portions of an image. Two special images exist, `WhiteImage` and `BlackImage`, which have data cards of their respective colors and are completely opaque.

All graphical operations in Montage are specified in a coordinate system that is one-to-one with pixels and has its origin in the upper left hand corner of the window to which the operation refers. Windows are clipped on display to their parent layer's rectangles, but offscreen portions may be drawn to without penalty, and will appear on the screen if the window is moved. Windows coordinates are relative to the upper left in order that windows do not have to be notified when they are moved.

The COOP Operator

The main graphics operator used in Montage is coop, for "compositing operator." A call to coop has the following form:

```
(coop dest-image dest-point
  source-image source-rect
  operation)
```

(The form of the call was intentionally modeled on the traditional monochrome `bitblt` operator.) The dest-point refers to the upper left of the rectangle affected within dest-layer; the size is determined from source-rect. The rectangles are clipped to lie within the boundaries of their respective layers. Figure 1 gives an example of the clipping that occurs in a coop operation; the grey rectangle indicates the parts of the source rectangle that are composited onto the destination.

Eleven compositing modes are defined, in addition to a separate operation which clears both the data and matte portions of an image. The coop operator is defined as a two-address instruction, where one operand, the source, is read and the other, the destination, is read and written. Nothing in the operator prohibits using a three-address variant, but the two-address form was picked for reasons of efficiency and simplicity.

Of the eleven operations, one (`D`) is a no-op and one (`S`) copies the source to the destination. `SoverD` is the image that contains the source, where its matte defines it as present, and the destination where it is present if the source is not. `SinD` contains the source where both the source and the destination are present, and nothing else. `SoutD` contains only the source where the destination is not present. `SatopD` consists of the parts of the destination that are not covered by the source and the parts of the source that cover the destination. `DoverS`, `DoutS`, `DinS`, and `DatopS` define the symmetric operations to the previous four. `DxorS`, which is the same as `SxorD`, contains the parts of the source and destination that are not covered by each other, and is clear both where they intersect and where neither is present.

The coop operations are illustrated in figure 2, using a grey triangle in the upper left as the source image and a black triangle in the upper right at the (initial) destination image. Both images are assumed to have mattes covering the regions that are not white. The grey image is implemented as a stipple pattern in the data portion of the layer, but the matte portion is not stippled.

Each coop operation is implemented as between one and four calls to a conventional `bitblt`[4, 10] routine, depending on what operation was specified and whether bitmaps or mattes are used for images.

Several other graphics primitives are implemented in Montage, for drawing standard objects such as lines, circles or text strings. All these operations work by specifying a color and a coop

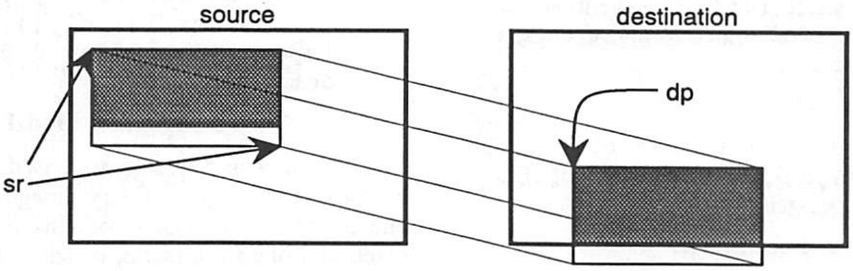


Figure 1. Clipping the coop operator.

operation in addition to the shapes described. For example, the `coop-string` procedure draws a string as if there were an image with a solid color data card and a matte containing a rasterized version of the string. Currently, only bitmapped fonts of horizontal orientation are supported.

Layers: Overlapping, Hierarchical, Translucent Images

The term "window" in Montage is reserved for the concept of a window that a user sees on the screen; windows are implemented in the underlying system by one or more "layers," which correspond to the concept of windows in X, canvases in NeWS, or layers in the Blit. Layers are organized as a forest of rooted trees; the screen is treated as a normal layer, named `*screen*`, at the root of one tree.

The children of a given layer are organized in a front-to-back order. Each layer has an associated priority, which defaults to 0 but may be set to any value in the range $-128 \leq n < 127$. Currently the only layer which has a non-zero priority is the cursor, which in all other respects is treated the same as any other layer. The same facility can be used, however, to provide effects like the "Icon Dock" in the NextStep user interface.[8]

There are three calls to create layers:

```
(make-layer parent rectangle)
(make-opaque-layer parent
  rectangle background)
(make-outline-layer parent
  rectangle color)
```

The rectangle specifies the area within the parent which the layer covers. While the child layer exists, drawing in that region of the parent is ignored. If the parent named evaluates to false, the new layer is created at the root of a new (offscreen) hierarchy.

`Make-opaque-layer` creates a layer with a solid (opaque) card as the matte; these layers are used for most clients, and it is the default type of layer created for a new application. Outline layers have cards for their data part, of the color specified in the creation command, and have bitmaps for their matte; they are similar to overlay canvases in NeWS, and are useful for rubber-banding a line, reshaping a window, or creating a grid for a draw program. `Make-layer` creates a layer with bitmaps for both data and matte; these layers can be used to create arbitrarily shaped windows (for round clocks or other special effects). The mouse cursor is created at system initialization with a make-layer request. At creation, opaque layers are textured with their specified background pattern; other layers are created with clear matte bitmaps.

Offscreen or obscured portions of layers have associated backing store, so programs never have to redraw windows. If portions of a layer are obscured, the entire image is stored in an offscreen bitmap. When a graphical operation is executed on an image in backing store, the operation itself is done only once, and the result of the operation are projection forward through the layer stack and up the hierarchy. If a layer is obscured by an opaque layer, the projecting stops at that point; when the obscuring layer is not opaque, the new image is computed with the `DoverS` compositing mode, and the projection continues.

While one might question the cost in memory size of this decision, experience has shown that applications which can count on the server maintaining a window's contents are significantly easier to

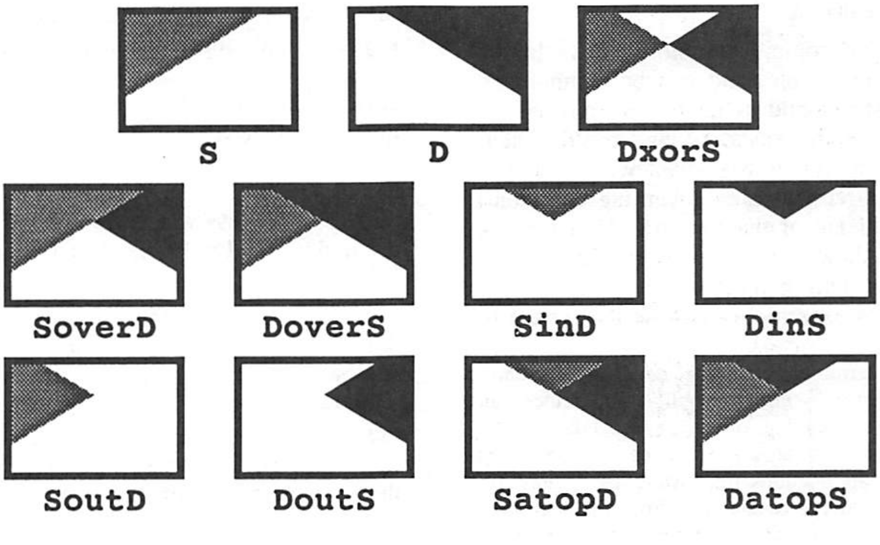


Figure 2. Compositing operations.

write. A window system that requires clients to redraw their contents after ''damage'' events is similar to a network protocol that requires applications to do their own error correction; while such services are usable, easy to implement, and well-suited to some specific applications (e.g., real-time video in a window or broadcasting a message), most programmers prefer to use a higher level service like reliable bytestreams instead of building their own error-corrected service on top of datagrams.

For more information about compositing, see Porter and Duff's paper.[13] Salesin and Barzel[16] discuss compositing in the context of monochrome displays. More details about the system used in Montage can be found in reference 5.

4. The Window System

The window system itself is constructed along the lines suggested by Pike.[11] All clients of the window system share a similar interface. Each client is started with a function call of the form

```
(client window keyboard
                  mouse co ci fd)
```

The window is a layer for the client to draw in. The keyboard is a channel on which a message is sent for every key press directed at the client. Similarly, the mouse channel receives a packet containing the current state of the buttons and a mouse coordinate every time either the buttons state changes or the mouse is moved. The channel `ci` is used for control messages from the window manager to the client; `co` is used for control messages initiated by the client. The control requests include reshaping or deleting windows. `Fd` is a connection to a peer process running on a host machine that implements the back end of the application; for example, in the case of a terminal emulator, the peer process would usually be a shell.†

The rules for communication over a client's channels are fairly simple, and can be summarized as: a client must respond as quickly as possible to messages on the mouse, keyboard, and control channels. Unlike the X Window System, where there is a large body of advisory rules governing communication between clients of the server,[15] Montage has few rules, but they are mandatory. Ignoring a request to quit or letting messages back up on the mouse channel, for example, can cause the system to lock up. In practice, maintaining the necessary invariants to keep the system running is not hard, but it is easier than it should be for one client to make the entire server unusable. One clear advantage that a window system, such as X, with a fixed client-server protocol and no client code running within the server has over a system like Montage is that a misbehaving client will rarely cause a correctly implemented X server to crash.

Each client is created by a window manager running within the server; the file descriptor is usually created in connection with the peer UNIX process of the window manager. There is a small protocol, available to Montage clients that want to create their own connections, which simulates creating a pipe shared by two processes that are connected by some connection. The protocol is implemented by having the process create a socket, tell the Montage server its hostname and port number, and listen on the socket. (If the two processes are on the same machine, a bidirectional pipe is used rather than an internet socket.) The Montage server connects to the socket and hands the file descriptor of the connection to the client.

In addition to creating clients, the window manager is responsible for multiplexing input among clients. It is in the window manager that user interface policy regarding input distribution is encoded. For example, two window managers have been written for Montage, one which implements a ''click-to-type'' user interface, the other a ''focus-follows-cursor'' policy. Changing interfaces requires only using a different manager. Each window manager is roughly 200 lines of concurrent Scheme, and they differ in only a few places.

The window manager itself has the same interface as its clients. Instead of being passed a child layer of the screen, its window is the screen itself. Similarly, at the other end of the window manager's keyboard and mouse channels are processes that read directly from the the input devices. The control channels for the window manager are connected to processes that let the window manager act as if it were running as a client of another window. The file descriptor for the window manager is connected to a UNIX process which forks child processes to run in windows and prepares their I/O connections to the server.

Pike has made the observation that if the window manager has the same interface as its clients, it can be used as a client as well as at the root of the window hierarchy.[11] One possible use for running the window manager recursively, is that an application which itself contains subwindows can delegate the responsibility for managing the windows to a new instance of the same window manager routine, called with slightly different parameters. Another use of running the window recursively is creating a subsidiary window manager in a different context

†The Montage terminal emulator does not actually communicate directly with a shell, but rather an intermediary process running on the same host as the shell. The problem with uses a direct connection arises because: (1) many applications run differently if they are invoked on a terminal (or pseudo tty) from the way they run on a pipe or network connection, so a pty is needed; and (2) a connection to a pty cannot be passed over the network.

from the root window manager, e.g., logged in to a remote machine or running as a different user.

It is precisely this feature which makes Montage convenient to use as a networked window system. The sequence to start a remote window manager in montage is quite easy: a user opens a new window, logins in remotely to another host, and gives the command `rws`. `Rws` (recursive window system) sends an escape sequence to the terminal emulator that causes the client part of the terminal emulator running within the Scheme interpreter to execute a fragment of Scheme code. The code fragment replaces the terminal with a window manager, which communicates with a window manager peer process that runs on the remote machine. After these steps (which appear to the user as logging in to another system and running one command), the window has become a new window manager. Inside the window, opening a new (sub)window causes a shell to start running on the remote machine; opening a new window from the root window manager still creates a local shell.

Figure 3 illustrates a window manager with three clients. Two of the clients are terminal emulators, the third is a recursive window system running on a remote machine. The lines with double arrows indicate pipes or TCP/IP sockets connecting Montage processes to UNIX processes. Lines with single arrows are drawn from UNIX processes to their child processes.

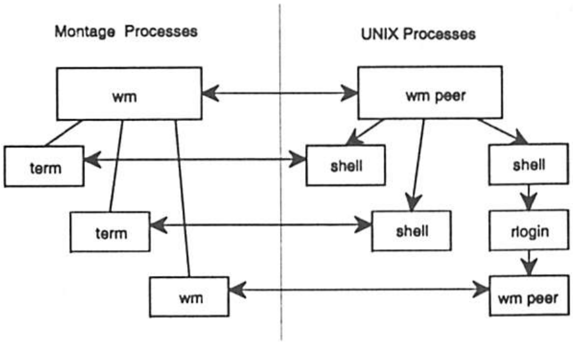


Figure 3: Configuration of processes

5. Objects, Widgets, and Processes

The most common technique for structuring window systems, and programmers' interfaces to window systems, is the object-oriented model. In fact, Smalltalk,[3] which is generally considered the first window system, is also the most consistently object-oriented programming environment in existence. Two aspects of the object-oriented model make it well-suited for constructing window systems: the object model forces programmers to identify and formalize interfaces; and the distribution of program state into objects, rather than a less-structured global context, hides some aspects of the asynchrony inherent in a window system from the programmer.

Montage consciously rejects the object-oriented model, using instead the multiple process model as the structuring paradigm. Processes in Montage are analogous to objects in object-oriented window systems. Things that would be objects in other systems, for example, the so-called ''widgets'' of X, are processes in Montage.

One can look at processes and objects as two different notations for the same concept. Both objects and processes associate local data with procedure, and both specify formal interfaces from outside to the private data: where objects provide this interface as part of a type system, processes specify a protocol to be used on their channels. The fundamental difference I would identify between objects and processes is that in an object, all state is explicitly maintained, whereas part of the state of a process (the location the ''program counter'' points to and the dynamic call stack) is maintained implicitly. This implicit state saving is a large part of the notational convenience behind using processes.

As an example, in an object-structured window system, unadorned windows would be one class, and windows with decoration would be a subclass inheriting from the plain windows. In Montage, a client, with the interface described above, can be wrapped in decoration with the following procedure.

`*decorate*` returns a client procedure with the same calling interface as the original, but the wrapped procedure has a thin border, that highlights when the window is focused, and responds to requests from the window manager to move to front or the back of the window hierarchy. This procedure implements a minimalist interface, similar to the one found on the Blit.[9] More complex intermediate processes are possible, and no part other than the wrapping code has to be aware of what is being done. In particular, the window manager knows no details of what `*decorate*` does, it just follows a predefined protocol, which is filtered by the decoration routine. (The reason that `tofront` and `toback` are handled in the decoration routine and not the window manager is that one decoration routine creates physically separate layersone for the window itself, one for a title barthat move to the front and back of the layer stack as a group.)

Other wrapping procedures are possible. One that has been implemented already is a keyboard abbreviation expander: the intervening process watches the keyboard channel for certain key sequencesthose with the meta key depressedand maps them to words from an association list. Again, the process that eventually receives the expanded key sequences does not know, and, in fact, can not determine, whether an abbreviated or expanded form was typed by the user.

A similar structuring can be applied to other user interface widgets, such as dialogue boxes, menus, scrollbars, etc. All a client of such a service is aware of is that a channel has at the other end of it a process which sends, say, selections from a menu. The anonymity of connections allows complex user interfaces to be built up from simpler ones, similar to the way UNIX pipelines are composed of independent filters.

6. Conclusion

Montage is an experiment in making window systems and their applications easier to write. The multi-process structure of Montage is a solid basis for constructing a window system, and provides a clean abstraction for implementing the services usually offered by a window system.

The graphics model used in Montage is perhaps its best feature. Using one data structure, the layer, for all visible objects is an effective unifying mechanism which gives the system a consistent feel for the programmer. The compositing algebra meshes extremely well with the idea of overlapping windows in a way that neither `bitblt` or PostScript[7] does.

Montage is not a practical system. The `call/cc` construct of Scheme, while powerful and portable, does not acheive the performance necessary for the near real time constraints that a window system imposes. Specifically, in every one of the dozen Scheme implementations that I have tested `call/cc` allocates so much memory that most of the run time is spent in the garbage collector. The Montage concurrency implementation provides the functionality one would want in a window system, but with performance only suitable for a prototype. For a language to support concurrency of the granularity used in Montage with adequate efficiency, it must be designed from scratch with concurrency in mind.

Acknowledgements

The work described in this paper was advised by Pat Hanrahan; Pat is also responsible for pointing me towards the compositing model. Dave Hanson advised my earlier work on window systems which led into Montage. Many of the ideas presented here were hashed out with Sean Dorward. Discussions with Norman Ramsey about his implementation of concurrency in ML influenced much of my own work. Matt Blaze, Sean Dorward, Pat Hanrahan, Pat Parseghian, and Byron Rakitzis provided comments

```
(define (*decorate* client)
  (define (decor $w $k $m $co $ci $fd)
    (let* ([lrect (layer-rect $w)]
           [border *decor-border*]
           [frect (rect-inset lrect border)]
           [w (make-opaque-layer $w
         (rect-inset frect (* 2 border)) White)]
           [ci(make-chan)])
     (define (intercept)
       (let ([mesg (<- $ci)])
          (case (car mesg)
          ((focus)
            (draw-border $w frect
              border Black SoverD))
          ((unfocus)
            (draw-border $w frect
              border White SoverD))
          ((tofront)
            (tofront $w))
          ((toback)
            (toback $w))
          (else
            (-> ci mesg))))
        (intercept))
     (draw-border $w lrect *decor-border* Black SoverD)
     (spawn (client w $k $m $co ci $fd))
     (intercept)))
 decor)
```

on drafts of this paper.

I would also like to thank Kent Dybvig and Cadence Research Systems for providing a copy of Chez Scheme for development of this system.

References

1. Luca Cardelli and Rob Pike, ''Squeak: A language for communicating with mice,'' *Computer Graphics* **19**(3) pp. 199-204 (1985).
2. R. Kent Dybvig, *The Scheme Programming Language,* Prentice-Hall, Englewood Cliffs, NJ (1987).
3. Adele Goldberg, David Robson, and Daniel H. H. Ingalls, *Smalltalk-80: The Language and Its Implementation,* Addison Wesley, Reading, MA (1983).
4. Leo J. Guibas and J. Stolfi, ''A language for bitmap manipulation,'' *ACM Transactions on Graphics* **1**(3) pp. 191-214 (July 1982).
5. Paul Haahr and Pat Hanrahan, ''If They're Called Windows, Why Can't We See Through Them?,'' Princeton University Computer Science Tech. Report (1990). (In preparation.)
6. C.A.R. Hoare, ''Communicating Sequential Processes,'' *Communications of the ACM* **21**(8) pp. 666-678 (August, 1978).
7. *NeWS 1.1 Manual.* January 1988.
8. *NeXT System Reference Manual, Version 0.9.* 1989.
9. Rob Pike, ''The Blit: a multiplexed graphics terminal,'' *AT&T Bell Labs. Tech J.* **63**(8) pp. 1607-1631 (1984).
10. Rob Pike, Bart Locanthi, and John Reiser, ''Hardware/Software Tradeoffs for Bitmap Graphics on the Blit,'' *Software -- Practice and Experience* **15**(2) pp. 131-151 (February 1985).
11. Rob Pike, ''A Concurrent Window System,'' *USENIX Computing Systems* **2**(2) pp. 133-153 (Spring 1989).
12. Rob Pike, ''Newsqueak: A language for communicating with mice,'' Computing Science Technical Report 143, AT&T Bell Laboratories, Murray Hill, New Jersey (January 1989).
13. Thomas Porter and Tom Duff, ''Compositing Digital Images,'' *Computer Graphics (Proceedings of SIGGRAPH 84)* **18**(3) pp. 253-259 (July 1984).
14. Jonathan Rees and William Clinger (editors), ''The Revised^3 Report on the Algorithmic Language Scheme,'' *ACM SIGPLAN Notices* **21**(12)(December 1986).
15. David S. H. Rosenthal, *Inter-Client Communication Conventions Manual,* MIT X Consortium Standard, Version 1.0 (1989).
16. David Salesin and Ronen Barzel, ''Two-Bit Graphics,'' *IEEE Computer Graphics and Applications* **6**(6) pp. 36-42 (June 1986).
17. R.W. Scheifler, J. Gettys, and R. Newman, *X Window System: C Library and Protocol Reference.* 1988.

Paul Haahr recently completed an AB in Computer Science at Princeton University. His research interests include window systems, programming language design, operating systems, and computer architecture. Haahr has previously worked at Oracle Corp., Belmont, CA, and Polygen Corp., Waltham, MA.

USENIX Summer Conference
June 11-15, 1990
Anaheim, California

swm: An X Window Manager Shell

Thomas E. LaStrange – Solbourne
Computer Inc.

ABSTRACT

swm is a policy-free, user configurable window manager client for the X Window System. Besides providing basic window manager functionality, **swm** introduces new features not found in existing window managers. First and foremost, **swm** has no default look-and-feel. Like the X Window system itself, **swm** does not dictate policy (look-and-feel); rather, it provides the mechanism for implementing window management policy. Users are not required to learn a new programming language to modify its behavior; instead, simple objects with associated actions determine **swm**'s operation. Its major advantage over other window managers is a feature called the Virtual Desktop. The Virtual Desktop effectively makes the X root window larger than the physical limits of the display and can be panned in a number of ways, including scrollbars, a panner object, or window manager commands. Besides window management, **swm** also provides primitive session management. It can save a user's current window layout and restart those clients when X is restarted. **swm** can restart clients regardless of what toolkit they were built on or what remote host (if any) they were running on. All relevant client information is restored, including window position and size, icon position, and the state of the client.

Introduction

Why another window manager? That is certainly a valid question given the number of window managers currently available for the X Window System. Current window managers fall into two categories: easy to use but not very configurable, or very configurable but complicated to use. Two currently available window managers that illustrate this point are **twm**[LaSt89] and **gwm**[Naha89]. **twm** is easy to use, but different window management policies are next to impossible to implement. **gwm** is policy-free, but requires command of the Lisp language to implement a particular look-and-feel. **swm** shares the best attributes of both of these popular window managers: it is very easy to implement a particular window management policy without the need to learn a new programming language.

swm is also a primitive session manager, a client that can save the current "session" of a user and restart that session sometime later. Users of SunView are familiar with the **toolplaces** program that allows users to move and resize windows and then save the current window layout such that each time they log in, the environment is remembered. This operation is much more difficult in the X Window environment for two reasons: first, client programs may have been built with different toolkits, each having a different set of command line options. Second, clients in the X Window environment are not constrained to be run on the same system that is actually running the X server; in fact, the system on which the client is running may not even be running the same operating system. **swm** solves both of these problems and saves the configuration information in a file suitable to replace the ".xinitrc" file commonly used to store X start-up information.

Architecture

swm is written in C++ using the Object Interface library (OI) [Aitk90]. As will be discussed in following paragraphs, **swm** is object oriented in that it deals with four basic objects to implement window manager appearance and behavior. The small number of objects used by **swm** helps reduce confusion for users. This object oriented approach is facilitated by OI because once a specific object is created, it can be treated as a generic base class object when dealing with attribute settings. This makes the interface to all objects consistent. It not only makes life easier for the user, but internally for **swm** also. When laying out panels, objects can be treated as base class objects without having to know the particular derived type of object actually being manipulated.

Configuration

All of **swm**'s configuration information is specified through the X resource database. This virtually eliminates the need for separate configuration files, such as the .twmrc file required by **twm**, and makes architecture-based configuration much easier. Because **swm** manages multiple screens on a multiscreen X server, there should be some easy way to specify resources on a per screen basis. All swm resources begin with the class of the window manager, either "Swm" or "swm", the latter having precedence. Following that, two strings indicate the screen number and whether or not the screen is monochrome. Some examples might include:

swm.monochrome.screen0
swm.color.screen1

This allows users to define the appearance or behavior of **swm** objects on a per-screen basis if desired.

swm resources fall into two categories: specific and non-specific. Specific resources are those associated with a client window such as an **xclock**. Non-specific resources deal with operational parameters of **swm**. Specific resources differ from non-specific resources in that both components of the WM_CLASS property of the client are included in the resource string. The syntax of specific and non-specific resources is:

Non-specific
  swm.*type.screen-number.resource*:
Specific
  swm.*type.screen-number.class.instance.resource*:

A full resource specification for the decoration of an **xclock** might look like this:

swm.monochrome.screen0.XClock.xclock.decoration:
  noTitlePanel

Using specific resources, one can specify different decorations, icons, behavior, and color for different application classes and instances of classes.

Several template files are supplied with **swm** to get the user up and running quickly. If no **swm** configuration resources have been specified, a default configuration can be loaded. In order to use one of the standard template files, a one line addition to the appropriate X resource file is all that is required. If a user wants to customize **swm**, they can either write their own configuration from scratch, or include and then override defaults in a standard template file. Among the template files are emulations for both the OPEN LOOK and OSF/Motif window managers.

Objects

swm deals with four basic objects: panels, buttons, text objects, and menus. From these four basic objects, an infinite number of window management policies can be implemented. Objects are arranged in hierarchies and can be stacked to any depth. Each object has its own set of attributes, such as color, font, and cursor. Each object also has a "bindings" attribute that describes the actions to be taken when mouse buttons or keyboard keys are used while the pointer is in the object.

Panel Object

The panel is the most basic object in **swm**. It is nothing more than a container for other objects. Objects within panels are organized into rows. A row within a panel is made up of one or more other objects. Panels can be divided into five distinct classes:

1. Decoration
2. Icon appearance
3. Root icons
4. Root panels
5. Icon holders

The syntax for defining a panel is the same regardless of what panel class is being created. The syntax specifies the objects within the panel and how they are positioned. The syntax for a panel definition is:

swm*panel.*panel-name*: \
  object-type *object-name* *position* \
  object-type *object-name* *position* \
  object-type *object-name* *position* \
  .
  .

panel-name is the name used to reference this panel. Each object within a panel has three components: *object-type* defines the type of object created. Valid object types include **panel**, **button**, and **text**. *object-name* defines the name used to reference the subcomponent; and *position* is a geometry string describing the position of the object within the panel. The X and Y components of the geometry map to the column and row position within the panel.

Decoration Panels

Decoration panels describe what client windows look like after they are reparented; they describe the "decoration" the window manager places around given client windows. The syntax for specifying a decoration panel is exactly like any other panel except that it contains an interior panel object called "client." For example, the definition below specifies the default decoration for a client window in the OpenLook+ template, which is supplied with **swm**. Figure 1 shows what a decorated window looks like using this panel definition.

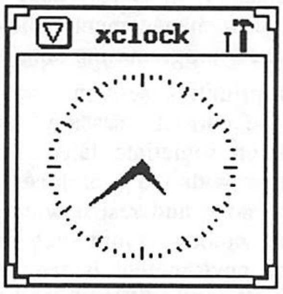


Figure 1: OpenLook+ Decoration

```
Swm*panel.openLook: \
  button  pullDown  +0+0 \
  button  name     +C+0 \
  button  nail     -0+0 \
  panel    client   +0+1
Swm*panel.openLook.resizeCorners: True
```

The openLook panel defined above is made up of four objects. In the above example, the button object called "name" has the letter "C" as its column coordinate, meaning to center the object within the row.

As noted earlier, the decoration panel must contain a panel object called "client." This special panel is where the client window is placed in the decoration panel. In the above example, the client window is placed in column 0 of row 1. Keep in mind that because the "client" panel is a generic panel object, there are no limits on what a decoration panel must look like. Objects can easily be placed to the sides or below the client window in addition to the more traditional "title-bar" appearance.

Another special object that can be used in a decoration panel is a button or text object called "name." This object displays the WM_NAME property of the client and can be seen in the above example. The "pullDown" and "nail" objects provide other functionality for this panel.

Icon Appearance Panels

swm has no concept of what an icon should look like; it is up to the user to describe how icons should be represented. Icon appearance panels serve this purpose. The default icon of the OpenLook+ template is defined as follows:

```
Swm*panel.Xicon: \
   button   iconImage   +C+0 \
   button   iconName    +C+1
Swm*defaultIconImage: @xlogo32
```

In the above example, the Xicon panel contains two buttons, both of which have special meanings. The iconName button displays the contents of the WM_ICON_NAME property. If the client has specified a pixmap to display as the icon or has specified its own icon window, that image is displayed in the iconImage button. In our example, if the client has not specified an icon, the iconImage button will contain the image of the xlogo32 bitmap file.

As with the decoration panels, there are no limits on how complex icon appearance panels may be.

Root Icons

The Icon Appearance panels described above are used when a corresponding client application has been placed in an iconic state. Root icons are simply icon appearance panels that do not correspond to any client application. Because of this they cannot be deiconified. Since they are icons, they can be moved, have bindings associated with them and can use icon holder panels (see below). What good are they? In particular, they can have bindings describing actions such as what should happen when they are the destination of an operation such as drag-and-drop.

Root Panels

Root panels are static panels (usually containing buttons) that are always visible on the root window. One can think of a root panel as a menu that is always visible. Root panels differ from Root icons in that they are treated like other client windows, i.e., they get reparented, can be iconified, etc. Figure 2 shows a reparented root panel and its definition:

```
Swm*panel.RootPanel: \
   button   quit      +0+0 \
   button   restart   +1+0 \
   button   iconify   +2+0 \
   button   deiconify  +3+0 \
   button   move     +0+1 \
   button   resize    +1+1 \
   button   raise    +2+1 \
   button   lower    +3+1
```

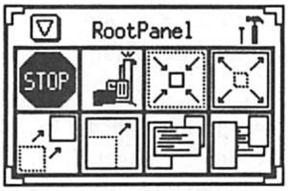


Figure 2: Root Panel Example

Icon Holder Panels

Icon holder panels are special root panels that contain icons. They provide an optional scrolling window in which icons can be deposited and managed. Icon holder panels are similar to the Icon Manager feature of **twm** but more flexible in that actual icons are managed rather than a fixed-appearance icon representation. Icon holder panels have many options, including being not displayed when empty, and sizing to fit all the icons rather than presenting a scrolling window. Icon holder panels can be created to contain specific classes of client icons. In this way you can do things such as group all **xterm** icons in one panel, and other icons in a separate panel.

Button Object

The button object can contain either text or a bitmap image. Buttons are heavily used in defining window manager decoration and icon appearance. The button object is unique in that its appearance can be changed dynamically through the use of window manager functions. This allows window manager decorations to change depending on the state of the client application. Like other **swm** objects, the button object can also have its bindings (functions) changed dynamically. With these two features, buttons can not only dynamically change appearance, but they can also change functionality.

Object Attributes

Each object, when created, queries the X resource database for a number of attributes. These attributes include color, font, cursor, and bindings. Among these attributes, the bindings attribute is the most interesting. The binding attribute defines the behavior of the window manager. Most other window managers, provide mouse and keyboard bindings in a particular context, i.e., window manager frame, icon, or client window. In **swm** you can think of each object as being its own context with its own set of actions. **swm** does not know that an object is within a window decoration panel or an icon, all it knows is that some object requested that a specific action take place when a mouse button or key event is detected within it. Object bindings are specified in an X Toolkit Intrinsics[McCo]90 format so that those familiar with the Xt syntax will not have to learn yet another way of specifying actions:

```
swm*button.foo.bindings: \
    <Btn1>    :  f.raise \
    <Btn2>    :  f.save f.zoom \
    <Key>Up   :  f.warpvertical(-50)
```

This example shows bindings for a button object called ''foo.'' Assuming that the button is in a window manager decoration panel, if mouse button one is clicked, the window is raised to the top of the stack. If mouse button two is clicked, the window's location and size are saved and the window is expanded to the full size of the screen. If the Up key is pressed while the pointer is over the button, the pointer will be warped ''up'' 50 pixels. It is useful to note that any number of bindings can be specified and that any number of functions can be specified per binding.

Window Manager Functions

All window managers provide functions allowing the user to do basic operations such as moving and resizing windows. In **swm**, functions are specified via a bindings attribute as shown in the previous paragraph. **swm** functions can be invoked in several modes. Using the **f.iconify** function as an example, here are the possible ways to execute the command:

f.iconify
: Iconify the current window

f.iconify(multiple)
: Iconify multiple windows, prompting for each window

f.iconify(blob)
: Iconify all windows whose class matches ''blob''

f.iconify(#$)
: Iconify the window under the mouse

f.iconify(#0x1234)
: Iconify a particular window ID

Another important feature of **swm**'s command execution is that commands can be executed on behalf of an outside client. By writing a special property on the root window, **swm** interprets its contents and executes commands. A simple client has been written called **swmcmd** that provides a way to execute window manager commands by typing them into a shell running in an **xterm**. This provides a mechanism for executing any command at any time; it is particularly useful when you have forgotten to add the command to one of your pop-up menus. By simply moving your pointer into any **xterm** and typing:

```
swmcmd f.raise
```

The pointer would be changed to a question mark prompting you to select a window to be raised. This interface could also be used for things such as changing the shape of a button to indicate the status of a process.

SHAPE Support

swm provides support for the SHAPE extension, allowing non-rectangular windows. Each object can have a separate shape mask attribute which is simply a bitmap image of the shape of the object. In addition to the shape mask, if a panel object is to be shaped and no shape mask is specified, it is shaped to contain its children.

swm recognizes if a client window is shaped and adds the string ''shaped'' to the beginning of the resource strings used to obtain attributes. This allows attributes to be specified for shaped client windows. A useful example would be to provide shaped decoration for shaped clients:

```
swm*shaped*decoration:  shapeit
swm*panel.shapeit:  panel  client  +0+0
swm*panel.shapeit*shape:  True
```

This particular decoration allows invoking the X11R4 **oclock** or **xeyes** clients and they would be displayed without visible decoration.

The Virtual Desktop

Probably the single most exciting feature provided by **swm** is the ''Virtual Desktop.'' This feature effectively makes the X root window larger than the physical limits of the display. This large ''root'' window can be panned using scrollbars, a two dimensional panner object, or window manager functions. Using the Virtual Desktop, it is very easy to implement a ''rooms'' like environment by grouping windows into various quadrants of the desktop.

Virtual Desktop Panner

The panner object provides several important features of the Virtual Desktop.

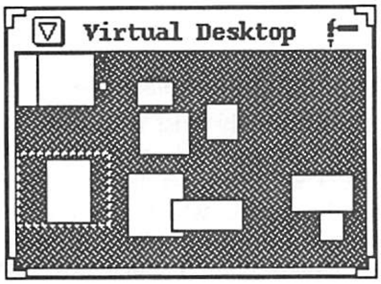


Figure 3: Virtual Desktop Panner

The panner shows a miniature representation of all windows currently on the Virtual Desktop. It also displays an outline indicating your current position within the desktop. When the pointer is in the panner and mouse button one is pressed, the current position outline can be moved to view another portion of the desktop. If mouse button two is pressed while the pointer is over one of the miniature windows, a move operation is started on the window. A small outline appears under the pointer representing the window shape that is being moved; it can be dropped anywhere in the panner and the actual client window is repositioned. If the pointer is moved out of the panner during the move operation, a full size outline of the window is displayed, allowing the user to move and fine tune the placement on the current visible portion of the desktop. This feature also works when the window move was started on a client window and the pointer is moved into the panner, allowing the user to move the client window to any portion of the desktop.

The panner is reparented so it can be moved, iconified, and resized just like any other client window. The act of resizing the panner object causes the underlying Virtual Desktop window to resize. This allows resizing the desktop at run time depending on the user's needs. Because the Virtual Desktop is an X window different from the actual root window, the size of the Virtual Desktop is limited only by the usable area of an X window, 32767 x 32767 pixels.

Sticky Windows

One might wonder why it is that the panner does not scroll off the display when the desktop is panned. In conjunction with the Virtual Desktop, **swm** introduces a feature called ''sticky'' windows. Sticky windows appear as though they are stuck to the glass of the display. When the Virtual Desktop is scrolled, sticky windows do not move. They allow the user to set up a standard environment that can be used regardless of the current position of the desktop. This standard environment might contain things such as a clock and mail notifier, which would then be visible no matter which portion of the Virtual Desktop is being viewed. Windows can be stuck and unstuck interactively by the user. Classes of windows can also be specified to start up as sticky and decorations can be dependent on whether or not the client window is sticky. Consider the following resources:

```
swm*xclock*sticky:  True
swm*sticky*decoration:  stickyPanel
```

These resources tell **swm** that any client that has the class ''xclock'' should start up as a sticky window. When **swm** finds that it is reparenting a sticky window, it places the string ''sticky'' in the resource string so that decoration and other attributes can be dependent on whether or not a client window is sticky.

Virtual Desktop vs. ICCCM

The Inter-Client Communications Conventions Manual (ICCCM)[Rose89] outlines a specific client/window manager communication protocol. The Virtual Desktop provides many challenges to both **swm** and clients in deciding exactly what should be done to remain ICCCM compliant.

Window Positions

Window positioning is probably the most common problem applications will encounter when running under **swm** with the Virtual Desktop enabled. Many clients monitor their position on the root window to allow intelligent positioning of dialog boxes or popup menus. When clients are moved to other portions of the desktop, they get positioning information which leads them to believe that they have been moved off the visible portion of the root window. When popping up a dialog box, the client may detect that its window is off the screen and attempt to position the dialog box back on the screen. This may cause it to be displayed in the upper-left quadrant of the desktop, possibly making it nonvisible.

Besides large position coordinates, other problems arise when the desktop is panned. If the desktop is positioned at 0,0, a window at location 100, 100 on the desktop is also at location 100, 100 relative to the real root window. If the desktop is panned to location 25, 25, the client window is still at location 100, 100, with respect to the ''root'' window onto which it has been placed, but is now at location 75, 75 with respect to the real root window. The window gets no ConfigureNotify events, real or synthetic, because it hasn't moved with respect to its root. If the client window is monitoring its position for use in popping up a menu or dialog box, its position with respect to the real root has changed and the popup window will probably be placed improperly.

swm and the OI toolkit have attempted to solve these problems. When **swm** reparents a window it places a property on the window indicating the window ID of its "root" window. This will be the window ID of the real root window or the ID of the Virtual Desktop window. The toolkit then reparents, maps, and positions popup menus and dialog boxes with respect to the window ID specified in the property rather than always using the actual root window. In addition, this property is updated whenever the root window for a client changes. This occurs, for example, when a window is made sticky or unsticky. Besides solving the window positioning problems, this would also allow **swm** to implement multiple Virtual Desktops, although at this point in time we're not sure how useful such a feature would be.

USPosition vs. PPosition

The ICCCM specifies two methods for X clients to request window positions: User Specified Position (USPosition) or Program Specified Position (PPosition). Both of these flags exist in the WM_NORMAL_HINTS property. These hints allow the user or application writer to request window positions that may or may not be honored by the window manager. The idea behind these hints is that if a user requested a window at a given location, it would presumably have priority over a default window location requested by the X client, which in turn might have priority over default window manager placement. Previous to X11R4, the X Toolkit Intrinsics always set the PPosition hint, effectively making the hint useless as all window managers simply ignored it.

swm uses these hints to determine window placement on the Virtual Desktop. If USPosition hints are specified, the window is placed at the absolute location requested by the user, even if the coordinates on the desktop are not currently visible. If PPosition hints are specified, the window coordinates are assumed to be relative to the current visible portion of the Virtual Desktop. For example, if the desktop is positioned such that the upper left corner of the display corresponds to pixel location 1000, 1000 on the desktop, a USPosition of +100+100 would place the window at 100, 100 on the desktop. If a PPosition of +100+100 is used, the window would be placed at 1100, 1100.

Because of the problems mentioned previously, multi-window applications wanting a default window layout usually set the USPosition hints to achieve their layout. While in most cases this presents no problems, it has the effect of making the application usable only in the upper-left quadrant of the Virtual Desktop, something many users will no doubt find objectionable.

Application writers needing a default window layout should make use of the PPosition hints and only use USPosition if the window positions have been requested by the user, either through the X resource database or through command line options. If compatibility with older window managers is desired, an option should be added to such applications so that the older behavior can be selected through a "bug compatibility" mode or something similar.

Session Management

There does not currently exist a standard mechanism in X for a user to save a "session" and then later restart the session. The most desired type of session management is one in which the locations of all client windows are saved, plus the state of each client. This would allow a window based editor to be shut down and brought up at some later date, editing the same files at the same positions within the files. Within the next few years this type of session management will gradually be designed, proposed, discussed, revised, and finally implemented. Until this happens, users have absolutely no standard way to do even basic session management, such as that provided by the SunView **toolplaces** program.

There are several reasons why even simple session management in X is difficult. First, since there is no standard toolkit, there are no standard command line options. The **xplaces** client attempts to do simple session management but assumes that X Toolkit Intrinsics options are used. This leaves users of the XView toolkit or other non-intrinsic based toolkits out in the cold. Second, client programs are not constrained to run on the same host as the X server and may even be using different operating systems.

swm attempts to solve both of these problems. It implements session management using a two step approach: first, an **swmhints** program provides **swm** with hints about the client's previous state; and second, **swm** interprets those hints to restore client windows to their previous state and location. The **swm** command **f.places** causes a file to be written which can be used as an .xinitrc replacement. This file contains two lines for each client to be restarted: an **swmhints** invocation with appropriate options, and the actual invocation of the client. The client is invoked with the exact command string found in the WM_COMMAND property. This allows restarting of clients regardless of the command line options they understand. An example client startup script might look like this:

```
swmhints -geometry 120x120+1010+359 \
  -iconGeometry +0+0 -state NormalState \
  -cmd "oclock -geom 100x100 "
oclock -geom 100x100 &
```

This example shows that the client **oclock** was originally started with a geometry of 100x100. Sometime later it was resized to 120x120 and positioned at location 1010, 359. If iconified, its icon would be positioned at 0, 0. Its state is NormalState as opposed to iconic. The **-cmd** option to **swmhints** simply echoes the WM_COMMAND property string looked for by **swm**. All of the information given to the **swmhints** program is appended to a property on the root window. When **swm** starts up, it reads this property and stores the information in an internal table. When a client window is reparented, the table is searched for a matching WM_COMMAND string and possibly a matching WM_CLIENT_MACHINE property. If a match is found, the entry is removed from the table and the window is sized and positioned to its old location. When restarting a client, **swm** restores the following attributes to the client window: window size, window location, icon location, whether or not the icon was on the root window, window sticky state, and the normal or iconic state of the window.

The scheme outlined above breaks down if two windows have identical WM_COMMAND properties. In this case, **swm** cannot distinguish between the two windows as they are being reparented. In practice this does not appear to be a problem for most users.

Restarting Remote Clients

Remote clients provide a challenge for session management. In a UNIX environment, if a user performs an **rlogin** to a system and then starts a client by simply typing the client name with no options, the client starts correctly if the DISPLAY, PATH, and any other environment variables needed by the client are set properly. In this case the WM_COMMAND property contains the client name and the WM_CLIENT_MACHINE property contains the name of the machine on which the client is running. This information by itself is not sufficient to restart the client. If the shell being used only reads an initialization file for login shells, it is very possible that the PATH and DISPLAY environment variables of the shell used for the remote startup will not be set or will not contain enough information for the client to run properly. **swm** provides the user with a resource that allows a customizable string to be used when starting remote clients. This string could be set up to include PATH and DISPLAY environment variable settings although this may limit portability when using the same startup file on different machines.

Evaluation

swm was developed, tested, and refined in approximately nine man-months. This relatively short development time was due to two factors: previous window manager design experience, and use of the OI Library. While this library provides the objects and interface to the window system that make it easy to write applications, there is certainly some performance penalty to be paid because of the extra overhead. **swm**, like any toolkit based window manager, has somewhat slower performance than a window manager written directly on top of Xlib or one that is kernel based. However, the added functionality and flexibility is well worth the speed trade-off, especially with the performance of newer machines.

One of the biggest mistakes made with **twm** was using a separate initialization file rather than the more general X resource database for configuration. The X resource database provides the user with many more configuration options, and makes extending **swm**'s capabilities easier.

Conclusion

In the introduction I posed the question, "Why another window manager?" The answer is relatively simple. The battle between user interfaces has not yet been won. **swm** was developed with this in mind. It is able to adapt to most any user interface. It also introduces new features, such as the Virtual Desktop, that are designed to improve user productivity. In addition, these features can all be configured by a standard method, the X resource data base.

References

[Last89] LaStrange, Tom, "An Overview of twm (Tom's Window Manager)," Xhibition '89 Conference Proceedings.

[Naha89] Nahaboo, Colas, "GWM, The Generic X11 Window Manager," Xhibition '89 Conference Proceedings.

[Aitk90] Aitken, Gary, "OI: A Model Extensible C++ Toolkit for X Windows," 1990 X Technical Conference proceedings.

[McCo90] McCormack, Joel, Asente, Paul, and Swick, Ralph, "Xt Toolkit Intrinsics - C Language Interface," X11R4 documentation

[Rose89] Rosenthal, David, "Inter-Client Communication Convention Manual," X Window System documentation

Tom LaStrange received a B.S. in Computer Science from the University of Utah in 1981. He got involved with X while working at Evans & Sutherland where he developed graphics microcode for the DEC VaxStation 8000 workstation to support X11R1. During his time at Evans & Sutherland, he also developed the initial version of the **twm** window manager for X. This window manager is now supported by MIT as the standard X window manager for X11R4. Following Evans & Sutherland, he worked for Hewlett Packard's Graphics Technology Division where he was involved in optimizing X server performance for HP's high-end graphics workstations. He is currently working in the User Interfaces and Applications group at Solbourne Computer, where he is involved in OI toolkit development and application design, as well as his work on swm. Reach Tom at 1900 Pike Road, Longmont, CO 80501. His electronic address is toml@Solbourne.COM.

USENIX Summer Conference
June 11-15, 1990
Anaheim, California

Michel Pedneault – Bell-Northern Research Ltd.

A High-Level User Interface Toolkit for the X WINDOW SYSTEM and Character Terminals

ABSTRACT

NTUI is a high-level user interface toolkit that allows single-stream development of UNIX applications intended for both the X Window System and character terminals. The graphical and the character-based versions of an application share the same source code. This sharing is not limited to code written in a high-level programming language such as C. It also includes defaults specified in X resource files, and objects, such as forms and menus, built with a user interface definition language that provides a non-programmatic way of specifying the presentation aspects of a user interface.

An important characteristic of this toolkit is that the character-based version does not impose constraints on the appearance of the graphical version. Similarly, the usability of the character-based version is not compromised by the richer interactive environment in which the graphical version is used. This is achieved by a careful encapsulation of appearance and behavior semantics at the toolkit level.

Both versions of NTUI are based on the X Toolkit. The X Window System version follows a client-server model. In the character-based version, windows and events are local to an application; a library provides windowing system and event-handling services that the X Toolkit normally obtains from an X Server. The sharing of a single terminal among several NTUI and other character-based applications is achieved by a screen manager which provides serial access to applications running in parallel.

Introduction

About two and a half years ago[*], a team of human factors experts was asked to design a state-of-the-art, graphical user interface style which would give Northern Telecom's products a clean, consistent, and professional image. This style, called NTUI, had to be particularly well-suited to the development of real-time applications. The resulting specification was implemented on several UNIX workstations in the X Window System [7] environment.

Today, however, most telecommunication products are operated from a character terminal. While these products would benefit from a direct-manipulation user interface, only a few really need the graphics capabilities offered by powerful, yet expensive, workstations. X terminals provide a cost-effective alternative to workstations, but according to analysts at International Data Corp. (IDC), the impact of X terminals on the character terminal market will not be felt until at least 1995 [1], and character terminals will still be part of the mix due to pricing considerations [2].

Although the first version of NTUI was graphical, the requirement to support character terminals had been identified early on. A character-based version of those applications that did not require a graphical display had to be available. Obviously, a graphical version also had to be provided in order to fully exploit the interaction and display capabilities of workstations and X terminals. Simply running character-based applications in an **xterm** window was not an acceptable alternative.

From a software management point-of-view, writing two versions of an application is highly undesirable. A partial solution is to separate the user interface of an application from its functionality; only the user interface then needs to be written twice. A much better solution is to specify the user interface with a high-level toolkit that shields the application from the environment in which it is used. The effort is then limited to the implementation of two device-specific versions of this high-level toolkit.

The goal of the high-level toolkit approach is to allow the graphical and the character-based versions of an application to share the same source code. This source code is not limited to code written in a high-level programming language such as C. In the

[*] October 1987.

X Window System environment, end users can control the appearance or function of an application with X resource files that specify default values for predefined resources. Some of these resources, like font and color, do not apply to character terminals, but some do. A resource used to specify the text to be displayed in a specific user interface component is an example of a resource that would apply to both environments. The integral support of X resource files in the character terminal environment is important because these files are maintained by end users.

In addition, NTUI and many other toolkits provide a specialized user interface definition language (UIDL) that allows for a declarative, rather than programmatic, specification of the presentation aspects of a user interface. Applications that do not require a graphical display and for which a character-based version has to be provided are essentially form-based applications: their user interfaces are built from predefined components assembled with a UIDL to create objects like forms and menus, while the high-level source code that deals with user interface issues typically concerns itself with getting and setting the value of attributes associated with individual components. To be useful, a high-level toolkit must allow the sharing of UIDL specifications.

When building a system that provides 100-percent source code portability between the X Window System and character terminals, one has to consider the fact that an X Server allows the sharing of a single display between several applications. Some subsystems are built as a collection of collaborating applications that rely on this screen multiplexing capability. As a result, some applications cannot be used independently of one another. In order to achieve portability at the subsystem level, it is necessary to extend the display sharing capability to character terminals.

Related Work

The Extensible Virtual Toolkit (XVT) [4, 6] is a high-level toolkit that provides portability between the X Window System and character screens. In addition, XVT provides support for the Macintosh, Presentation Manager, and Motif environments.

XVT is a thin layer of software that implements a common application programming interface across the supported environments. XVT calls map to lower-level native toolkit calls. Applications created with XVT have the look and feel of applications created with the native toolkit.

XVT abstracts the properties that the supported systems share. Although XVT avoids the least common denominator approach by providing an interface "that represents what application programmers need to do" [5], it tends to be constrained by each one of the supported toolkits. Direct calls are supported for applications needing access to the native toolkit, but the resulting applications are nonportable.

XVT allows the user interface objects for the various environments to be specified using a Universal Resource Language (URL). A URL file is compiled to produce a UIDL file for the target environment. This allows portability of the user interface objects as well as the application code.

XVT provides a unified way of writing to windows, and of handling mouse and keyboard events. This is useful when writing portable graphical applications, but, in order to remain portable between graphical and character-based environments, form-based applications should leave the handling of raw events and screen updates to predefined user interface components.

High-Level Toolkit

The NTUI style is implemented as a high-level toolkit that shields an application from the environment in which it is used. The two implementations of the NTUI toolkit define a common set of user interface components. Their appearance and behavior are adapted to the device on which they have to be displayed and used. UIDL objects built from these components can be used in both environments.

The two versions of the NTUI toolkit provide the same application programming interface. Once an application has been compiled, it simply needs to be linked with the appropriate set of toolkit libraries to produce a device-specific executable.

Policy, Consistency and Style

First and foremost, NTUI is a user interface style. The style is concerned with both the *look*, which defines how the interface is displayed, and the *feel*, which defines how the interface is used. This user interface style is realized by a high-level toolkit that provides a suite of predefined user interface components that can be assembled to build a user interface. Both versions of the NTUI toolkit are based on the X Toolkit [3], a mostly policy-free foundation on which a wide variety of styles can be implemented. To the extent possible, the NTUI toolkit implements and enforces a consistent style across applications.

For example, the NTUI style specifies that a scrollbar must be shown when a multi-line text field contains more text than can be displayed. The slider position and size in the scrollbar show the relative position and size of the currently displayed text. This means that every time a character is inserted or deleted, the scrollbar's slider must be updated. Conversely, every time the slider is moved, the text must be scrolled to reflect the new slider position. In NTUI, this style is implemented by the scrolling text field widget. This widget creates a text field widget and a vertical scrollbar widget, and it manages the interactions between these two children. An application programmer does not have to write a single line of code to implement this look and feel. There are many instances of high-level widgets like the scrolling text field in NTUI. Their purpose is to help enforce policy, but they also speed up user interface development (and this is probably as important).

Currently, the NTUI toolkit consists of 75 widget classes ranging in complexity from a simple static text widget to a powerful chart widget. Twelve of these widget classes are used as supporting superclasses for other widget classes. An application can directly create widget instances for 26 high-level widget classes. Instances of the remaining 37 widget classes are created indirectly when creating high-level widgets. Twenty of these widget classes are hidden from applications. Instances of the remaining 17 widget classes are called subwidgets. In the previous example, the scrolling text field was a high-level widget, the vertical scrollbar was a subwidget, and the text field was a hidden widget. Having the vertical scrollbar as a subwidget of the scrolling text field is a way of enforcing the NTUI style. A vertical scrollbar will be created, and displayed whenever it is required.

Even though an application cannot create subwidgets directly, it can get and set the value of attributes associated with subwidgets. It can also influence how subwidgets are created by specifying a list of arguments to use when creating them. For example, when creating a scrolling text field widget, an application can set an attribute on the vertical scrollbar subwidget to specify a list of procedures to execute when the menu associated with the scrollbar is about to pop up. However, the position and size of the scrollbar's slider can only be set by the scrolling text field widget. Hidden widgets like the text field widget transparently inherit attributes from their parent.

The extensive use of subwidgets and hidden widgets to enforce policy greatly facilitates the successful mapping of a graphical, direct manipulation user interface to a character terminal. The toolkit prevents invalid combinations of user interface components. As the number of valid combinations is limited, the number of problems that widget writers have to solve is finite, and each one of them can be tackled individually. Also, as the toolkit specifies how individual user interface components are displayed and used, the toolkit designers can adapt appearance and behavior to suit a specific environment. The use of subwidgets and hidden widgets extends this fine-tuning capability to collections of components.

Display Semantics Encapsulation

The appearance of a character-based widget has to convey the same meaning as its graphical equivalent. In order to achieve this, the display semantics have to be encapsulated in the widget. An example of such an encapsulation is found in the window widget. To indicate the highest level of alarm condition associated with the task represented by a window, an application can set the **status** attribute of the window widget to **statusCriticalAlarm, statusMajorAlarm, statusMinorAlarm,** or **statusOk.** In the graphical version of the window widget, a predefined icon is displayed based on the value of the status attribute. In the character-based version, a string is displayed instead of an icon. Similarly, widgets that allow an icon to be specified also allow the application to specify an alternate string representation of the icon's meaning. For example, the graphical version of the raster widget displays an uneditable raster image, but the character-based version of this widget displays an uneditable character string.

Two high-level NTUI widgets do not have a character-based implementation. One of them is the graphics work area widget, which provides an application with a widget in which arbitrary graphics operations can be performed. The other is the chart widget, which provides an application with the means for displaying data in a graphical form. The character-based version of the toolkit simply defines them so as to provide the same application programming interface as the graphical version.

Each version of the toolkit also provides a procedure that an application can call to determine in which environment it is being used. An application could, for example, use this information to decide whether a data set should be presented as a pie chart or as a table.

Behavior Semantics Encapsulation

In a graphical environment, the user can use both the mouse and the keyboard to operate the interface. It might seem, at first sight, that this rich interactive environment would have to be emulated in the character terminal environment. However, this would only be true if an application were allowed to assume that a mouse had to be used to invoke certain operations, and if that application extracted information from mouse events to determine how these operations should be performed. Fortunately, the X Toolkit makes the task of encapsulating the semantics of user actions relatively easy for a widget writer.

The X Toolkit delivers events to widgets, not to applications. Actions can be defined on a widget, and a translation table specifies the actual sequence of keys or mouse events that will trigger an action. Whether a mouse or key event caused the action to be invoked is only important for the widget writer, not for the application programmer. But in order to properly encapsulate user actions at the widget level, the widget writer has to be careful not to simply relay them to applications. For example, an application should only be notified when all the criteria to activate a menu item are satisfied. The application should not be involved in all the steps required to activate the menu item.

User interface mechanisms that work well in a graphical environment cannot blindly be transported to a character-based interface. When a user interface primarily intended to be operated by direct manipulation has to be used without a mouse, special care must be given to how end-users select and manipulate objects. In addition, a host and a terminal normally communicate using a relatively slow connection. In order for a user interface to remain usable, the number of times the screen is updated, as well as the number of characters sent for every update, have to be minimized. The screen size of a character terminal is also limited. A window is often created with a smaller than optimum size, causing certain objects to be hidden. The end-user then has to scroll the contents of the window to get access to these objects.

For example, when tabbing from object to object, the cursor might be moved to an unviewable object. One option is to automatically scroll the object into view. But if the user's intent is to navigate to another object, refreshing the screen at this point would slow things down. When such a situation occurs, the character-based version of the NTUI toolkit moves the cursor to a scrollbar to show that scrolling is required to view the object. The user can then tab to another potentially unviewable object. Alternately, the display can be scrolled, and the cursor is positioned on the object as soon as it becomes viewable. Finally, if an action is performed on the unviewable object (for example, typing text in a data entry field), it is automatically scrolled into view.

X Server Surrogate

The two libraries and the application described in this section are used to replace functions normally provided by an X Server with character terminal substitutes.

Character Terminal Base

The character terminal base is a library that consists of a curses-based windowing system which provides overlapping, hierarchical, resizable, rectangular windows owned by a single application, and an event handler which provides access to a queue of keyboard- and window-related events similar to those generated by an X Server.

The character terminal base was built so as to allow NTUI applications to be accessed from a large variety of character terminals. To achieve this, it relies on a table-driven definition of the terminal. The display characteristics of a given terminal are extracted from terminfo, a terminal capability database found in many UNIX environments. To solve the keyboard input aspect of the problem, the character terminal base defines a virtual keyboard. The keys and key modifiers defined on the virtual keyboard match the standard set of key symbols (for example, **Help**) and key modifiers (for example, **Meta**) defined by the X Window System. A keyboard database describes how different physical keyboards can be used to emulate the virtual keyboard. Entries in the keyboard database define how a key or sequence of keys on a physical keyboard is mapped to a potentially modified key on the virtual keyboard.

Xlib Substitute

In the graphical version of NTUI, the X Toolkit calls procedures defined in the standard Xlib library [8] to operate on windows and extract events from the input queue. The substitute Xlib library used in the character-based version provides the same application programming interface as the standard Xlib library, but instead of communicating with an X Server, it calls procedures defined in the character terminal base library.

The X Toolkit also calls resource manager procedures defined in the Xlib library to get system and user defaults. These procedures are also supported in the substitute Xlib library. This allows X resource files to be supported as is.

Screen Manager

The purpose of the screen manager is to give a user the ability to run several interactive applications on the same character terminal. The current implementation allows serial access to applications running in parallel. Each application runs in its own virtual screen, and a user can see only one of them at a time. Except for key sequences used to navigate from screen to screen, keyboard input is redirected to the visible application. Each virtual screen is implemented as a pseudo-terminal that emulates a DEC VT100. The screen manager saves and updates the contents of every virtual screen; applications running in background screens can continue to generate output as if they were in full control of the physical screen.

Screen management is done transparently from applications. An application does not need to do anything special to run with the screen manager. This allows third-party applications like **csh** and **vi** to be supported. On the other hand, the screen manager provides NTUI applications with value-added functions. A user can cut and paste textual information between NTUI applications. The screen manager also arbitrates requests to get exclusive control of the physical screen. The NTUI toolkit makes such a request on behalf of an application when a modal dialog is popped up. Modal dialogs are used to display extremely urgent information that requires immediate attention and response. If another application already has exclusive control of the physical screen, the request is queued. Otherwise, the modal dialog is displayed, and navigation to other screens is disabled until the modal dialog is dismissed.

Examples

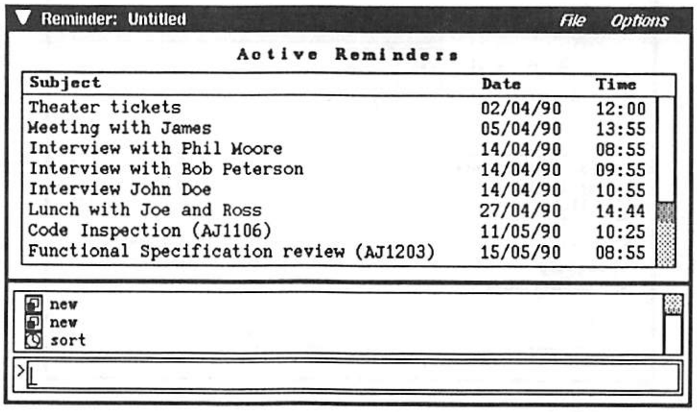
NTUI applications usually have a permanent window, called the anchor window, that is the focal point of interaction with the user. An anchor window is created using a window widget. This widget consists of a titlebar, a work area, and an optional command pane. The titlebar is displayed above the work area.

When the command pane is present, a pane separator is displayed between the work area and the command pane. Any increase in the size of the command pane reduces the size of the work area. In the graphical environment, the pane separator can be moved with the mouse to reallocate space between the command pane and the work area. In the character-based environment, grow or shrink key sequences can be entered from any widget to resize the pane in which the widget is located.

Figure 1 shows the anchor window of the graphical version of **Reminder**, a sample NTUI application that allows the creation of reminders that are displayed at a specified time. Figure 2 shows the anchor window of the character-based version of **Reminder** running in an **xterm** window. The same source code was used to produce both versions of the application.

The window widget makes an extensive use of subwidgets. The command pane and the pane separator are subwidgets of the window widget. The menus that can be associated with a window widget are also defined using subwidgets. One of these menus is called an object menu. It contains menu entries that are used to operate on the associated widget. For example, the object menu of **Reminder**'s anchor window contains an entry that can be used to close the window.

Most NTUI widgets and subwidgets can have an object menu. Every menu entry in an object menu is a subwidget of the associated widget. The **objectMenu** attribute of a widget defines its object menu. The data structure specified as the value of this attribute allows a name, and an argument list to be specified for each menu entry.



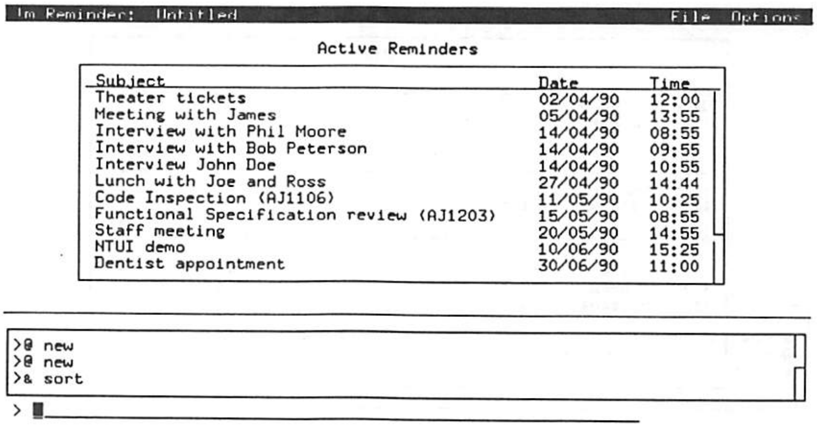
```
                          Active Reminders

Subject                                      Date       Time
Theater tickets                              02/04/90   12:00
Meeting with James                           05/04/90   13:55
Interview with Phil Moore                    14/04/90   08:55
Interview with Bob Peterson                  14/04/90   09:55
Interview John Doe                           14/04/90   10:55
Lunch with Joe and Ross                      27/04/90   14:44
Code Inspection (AJ1106)                     11/05/90   10:25
Functional Specification review (AJ1203)     15/05/90   08:55

new
new
sort

>
```

Figure 1: Graphical version of a sample NTUI window.



```
                          Active Reminders

 Subject                                      Date       Time
 Theater tickets                              02/04/90   12:00
 Meeting with James                           05/04/90   13:55
 Interview with Phil Moore                    14/04/90   08:55
 Interview with Bob Peterson                  14/04/90   09:55
 Interview John Doe                           14/04/90   10:55
 Lunch with Joe and Ross                      27/04/90   14:44
 Code Inspection (AJ1106)                     11/05/90   10:25
 Functional Specification review (AJ1203)     15/05/90   08:55
 Staff meeting                                20/05/90   14:55
 NTUI demo                                    10/06/90   15:25
 Dentist appointment                          30/06/90   11:00

>@ new
>@ new
>& sort

> █
```

Figure 2: Character-based version of a sample NTUI window.

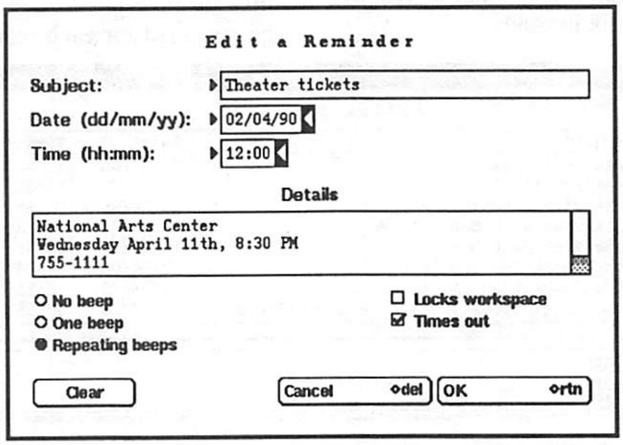
The titlebar displays the **status**, and the **title** of an application. In figure 1, the **status** of the window is displayed using the predefined **statusMinorAlarm** icon. In figure 2, the string "!m" is displayed instead of an icon. The titlebar also displays the title of tool menus. The object menus associated with these subwidgets contain menu entries that are used to operate on the contents of a window. In **Reminder**, for example, the *Options* tool menu contains an entry that can be used to hide or show the command pane.

The command pane subwidget consists of a command field, where the user types in commands, and an optional history field, which displays previously entered commands. Commands can be copied from the history field to the command field. Both the history field and the command field are subwidgets of the command pane.

The window widget can be the parent of only one widget. This child defines the contents of the work area. In **Reminder**'s work area, a static text widget and a list widget have been laid out using a form widget. This form is the child of a viewport widget that automatically provides scrollbars when the work area is too small to display all the contents of the form widget.

The initial size of the anchor window shown in figure 1 was set to the size required to satisfy the optimal size of the work area and of the command pane. The optimal size of the work area was computed based on the preferred size of the list widget as specified by the **heightInLines** attribute. In the character-based version of the NTUI toolkit, anchor windows are always sized so as to be the same size as the screen. In figure 2, for example, we can see that the form widget made the list widget grow vertically in order to take advantage of the extra space.

The form widget allows an application to specify the initial size and location of each child in the form, and how children should be resized and repositioned when the form is resized. For example, a child can be placed at a specified distance below another widget in a form. If the form is resized, the child will remain below the reference widget
even if the reference widget grows. An example of a flexible layout is provided by the graphical (figure 3) and the character-based (figure 4) versions of one of the dialogs associated with **Reminder**. The form



Edit a Reminder

Subject: Theater tickets

Date (dd/mm/yy): 02/04/90

Time (hh:mm): 12:00

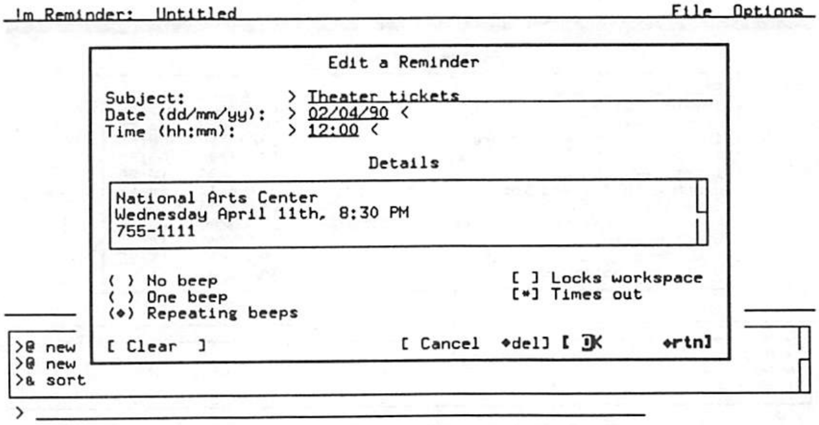
Details

National Arts Center
Wednesday April 11th, 8:30 PM
755-1111

○ No beep
○ One beep
● Repeating beeps

□ Locks workspace
☑ Times out

Figure 3: Graphical version of a sample NTUI dialog.



```
!m Reminder:  Untitled                                        File  Options

                          Edit a Reminder

    Subject:              > Theater tickets
    Date (dd/mm/yy):      > 02/04/90 <
    Time (hh:mm):         > 12:00 <

                              Details

    National Arts Center
    Wednesday April 11th, 8:30 PM
    755-1111

    ( ) No beep                              [ ] Locks workspace
    ( ) One beep                             [*] Times out
    (◆) Repeating beeps

>@ new
>@ new   [ Clear  ]                 [ Cancel  ◆del] [ OK        ◆rtn]
>& sort

>
```

Figure 4: Character-based version of a sample NTUI dialog.

layout looks natural in both environments.

Dialogs are transient windows used to display messages or gather information. Unlike character-based anchor windows, character-based dialogs are not constrained to be the same size as the screen. However, these dialogs cannot be bigger than the screen and are not allowed to extend past the screen boundaries.

The relative or absolute position of some of the widgets shown in figure 4 was specified in pixels. The character-based version of the NTUI toolkit treats values that specify a horizontal position as 1/8th of a column, and values that specify a vertical position as 1/16th of a row. This approach allows a widget layout specified in pixels to be used on a character terminal. None of the widgets shown in figure 4 had its horizontal or vertical dimension specified in pixels, but the same technique would have been used to transform these values.

Conclusion

NTUI provides a cost-effective solution to the problem of having to support both a character-based and an X Window System version of a UNIX application. High-level source code, X resource files, and UIDL objects do not have to be rewritten or recompiled. The application simply needs to be linked with the appropriate set of libraries to produce a device-specific executable.

NTUI is implemented as a high-level toolkit that shields the application from the environment in which it is used. The risk with this approach is that both versions of the toolkit could turn out to be unsatisfactory. The limited display capabilities of character terminals might impose constraints on how graphical user interface components are displayed. Similarly, the rich interactive environment provided by the X Window System might make the character-based version hard to use. The NTUI toolkit avoids these problems by a careful encapsulation of appearance and behavior semantics at the toolkit level.

Both versions of NTUI are based on the X Toolkit. This allows a significant amount of code to be shared between the two versions; some widgets are even identical. The X Toolkit was found to be particularly helpful with the encapsulation of user actions.

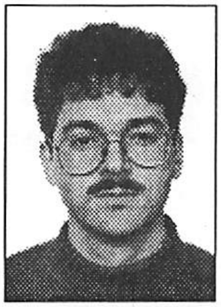
The screen manager extends the display sharing capability provided by X Servers to character terminals. This allows subsystems built as a group of collaborating form-based applications to be ported to character terminals. The screen manager also supports third-party applications like **csh** and **vi**.

Acknowledgements

The author would like to thank the designers of the NTUI style, especially Ross Holden and Miguel Planas, and the members of the NTUI implementation team, especially James Doucette, Steven Armstrong and Johanne Davidson. Thanks also to Joe Hatz, Jon Meads, and Michael Schwarz who provided helpful comments that improved the presentation of this paper. Special thanks to Jules Meunier for being a man of vision.

References

1. H. Baldwin, Why all the Shouting Over X Terminals, *UNIX World, Networking Supplement*, 75-81, 1989.
2. J. Brunet, Using PCs as X Servers, *UNIX World, Networking Supplement*, 83-85, 1989.
3. R. Rao, S. Wallace, The X Toolkit, the Standard Toolkit for X Version 11, *Proceedings of the Summer 1987 USENIX Technical Conference*, 117-129, 1987.
4. M. J. Rochkind, XVT: A Virtual Toolkit for Portability Between Window Systems, *Proceedings of the Winter 1989 USENIX Technical Conference*, 151-163, 1989.
5. M. J. Rochkind, *Technical Overview of the Extensible Virtual Toolkit (XVT)*, Boulder, Colorado: Advanced Programming Institute Ltd., 1989.
6. M. J. Rochkind, A Unified Programming Interface for Character-Based and Graphical Window Systems, *Proceedings of the Summer 1989 USENIX Technical Conference*, 109-117, 1989.
7. R. W. Scheifler, J. Gettys, The X Window System, *ACM Transactions on Graphics*, 5(2):79-109, April 1986.
8. R. W. Scheifler, J. Gettys, R. Newman, *X WINDOW SYSTEM - C Library and Protocol Reference*, Bedford, Massachusetts: Digital Press, 1988.



Michel Pedneault is a Member of the Scientific Staff at Bell-Northern Research. He joined BNR in 1986, after receiving his M. Sc. in Computer Science from the University of Montreal. His current interests include windowing systems, user interface toolkits, and user interface management systems. Write to Michael at P.O. Box 3511 Station C; Ottawa, Ontario; Canada K1Y 4H7. Send electronic mail to him at pedneaul@bnr.ca or uunet!bnrgate!bmers16!pedneaul.

Acknowledgements

[illegible] would like to thank the designers of the [illegible] and Miguel [illegible] of the NTIS implementation [illegible] Steve Cannon [illegible] helpful comments that improved the presentation of this paper. [illegible]

References

[illegible]

[illegible] is a Member [illegible] Staff at Bell[illegible] Research. He joined [illegible] in Computer Science from the University of Minn[illegible] [illegible] Ph.D. in [illegible] Canada [illegible]

[illegible]

USENIX Summer Conference
June 11-15, 1990
Anaheim, California

The LFS Storage Manager

Mendel Rosenblum, John K. Ousterhout –
University of California, Berkeley

ABSTRACT

Advances in computer system technology in the areas of CPUs, disk subsystems, and volatile RAM memory are combining to create performance problems existing file systems are ill-equipped to solve. This paper identifies the problems of using the existing UNIX file systems on 1990's technology and presents an alternative file system design that can use disks an order-of-magnitude more efficiently for typical UNIX workloads. The design, named LFS for log-structured file system, treats the disk as a segmented append-only log. This allows LFS to write many small changes to disk in a single large I/O while still maintaining the fast file reads of existing file systems. In addition, the log-structured approach allows near instantaneous file system crash recovery without coupling CPU and disk performance with synchronous disk writes. This paper describes and justifies the major data structures and algorithms of the LFS design. We compare an implementation of LFS in the Sprite distributed operating system to SunOS's file system running on the same hardware. For tests that create, destroy, or modify files at a high rate, LFS can achieve an order-of-magnitude speedup over SunOS. In spite of its obvious write-optimization, LFS's read performance matches or exceeds the SunOS file system under most common UNIX workloads.

Introduction

When the current UNIX file systems were designed, a ''fast'' computer had a one-MIPS CPU and at most a few megabytes of memory. The machines of the 1990's will have many hundreds of MIPS of CPU and many hundreds of megabytes of memory. Unfortunately, the disks attached to these machines will likely have access times within a factor of five of the 1970's disks. Unless current file systems are modified to match this change of balance between technologies, the machines of the 1990's will operate in an I/O-bound mode where order-of-magnitude increases in CPU speed may produce no visible speed-up in applications.

As faster CPUs shrink the CPU time of programs, the overall execution time will be dominated by the time to perform disk accesses. Large volatile main memories will permit large file caches. Disk delays due to file reads will be alleviated by these caches but file writes must be pushed to disk for reliability. Disk traffic will be dominated by these writes. The challenge for computer systems of the 1990's will be to support these workloads without requiring the redesign of programs in order to change their file access patterns. For the UNIX environment this means efficiently supporting changes to many small files as well as large files.

In the existing UNIX file system, creation and deletion of small files cause disk accesses that are small, non-sequential, and synchronous (the application cannot continue until the disk I/O completes.) The synchronous accesses limit application performance to disk performance. The small non-sequential accesses limit the disk bandwidth utilization to a fraction of the disk's maximum bandwidth. File systems of the 1990's must decouple application and disk performance and use the disks much more efficiently to reduce the I/O bottleneck.

This paper describes a disk storage manager designed to use disks as efficiently as possible to support write-dominated workloads. The storage manager, called **LFS**, uses the concepts of **log-structured file systems**[1] to increase the performance of the UNIX file system. In a log-structured file system, all modifications to the file system including data, directories, and metadata blocks are written to disk in large, sequential transfers that proceed at maximum disk bandwidth. Small file creation and deletion in LFS cause disk accesses that are large, sequential, and asynchronous.

LFS's large sequential writes cause a disk layout that is very different from existing UNIX file systems. Although the layout differs, LFS maintains many of the same metadata structures such as inodes and indirect blocks. This allows LFS to efficiently support the full UNIX file system semantics including fast random and sequential read access to files. The major difference between LFS and UNIX disk layout is that inodes, the disk resident data structure containing file attributes, are not at fixed locations on disk in LFS. LFS must maintain a data structure that tracks the current location of the inode of every file. Once LFS has indexed through this data structure, file reads are identical to UNIX.

The log-structure of LFS requires that large disk regions be available for writing. LFS manages these regions by partitioning the disk into large sequential pieces called **segments.** Blocks are added to the file system a segment at a time and disjoint segments are linked together to form a logically consecutive segmented log. In order to keep segments available for writing, LFS performs an operation called **segment cleaning.** The operation reads fragmented segments into memory, compacts the live data, and writes it back to segments on disk.

The log-structure of LFS provides several other benefits in addition to high performance. The log allows LFS to recover from system crashes much faster than existing UNIX file systems. LFS never needs to scan the entire file system to recover from a crash. The append-only nature of LFS means that files and directories are not updated in place. This allows LFS to provide better crash guarantees and higher performance.

The rest of this paper comprises 5 sections. It begins with an examination of current technology trends and their effects on storage manager design. The following sections describe failures of existing file systems, the LFS design, and some performance numbers taken from an implementation in the Sprite distributed operating system[2]. The paper concludes with status and future directions for LFS.

Technology and storage managers

This section examines the effect of current technology trends in CPU speeds, disk performance, and main memory sizes on storage managers. Many of the design decisions of LFS can be attributed to the desire to allow graceful scaling of systems in the face of unbalanced technology changes.

Disks and CPUs speed

The most obvious technology mismatch affecting disk storage managers is the exponential increase in CPU speeds compared with the small increase in disk access speeds. The later part of the 1980's has seen CPUs speeds doubling every year. Disk access times, constrained by mechanics, have only improved by a factor of two in the last decade. Although the the bandwidth and throughput of disk subsystems can be substantially increased by the use of arrays of disks such as RAIDs[3], the access time for small disk accesses is not substantially improved and can even be hurt by this technique. The widening gap between CPU and disk access time suggests that the performance of future systems may be limited by the disk subsystem.

Memory sizes and disk read/write ratios

Disk and CPU speeds are not the only technology that affects storage manager design. Main memory sizes of machines are increasing with CPU speeds. The fast CPUs of the 1990s will have multi-billions of bytes of memory, from which effective file caches will be built. Large file caches will alter the read-to-write ratio presented to the disk subsystems. Most reads will be satisfied by the cache while most writes must be written to disk for reliability. In contrast to the file systems of the 1970's and 1980's, in which disk traffic was dominated by reads, the file systems of the 1990's will see disk traffic dominated by writes.

Storage manager design

The radical difference in CPU and disk speeds make it imperative that storage managers decouple an application's performance from the disk access speed. This section describes disk access modes that storage managers can exploit in order to use disks more efficiently.

Asynchronous I/O

Storage managers can lessen the effect of the disk/CPU speed unbalance by avoiding operations in which the CPU is forced to wait for disk operations to complete. Blocking operations, called synchronous disk operations, couple the CPU and disk by requiring the requesting processes to wait while the disk is accessed.

Sequential I/O

Disk access efficiency depends on how the disk is accessed. Disk subsystems can be accessed in sequential mode an order-of-magnitude more efficiently than they can be accessed randomly. Storage managers should exploit this property by performing large sequential accesses. Disk head motion (seeks) should be avoided whenever possible.

Problems with existing UNIX file systems

Before presenting the design of LFS it is useful to outline the failures of existing file systems. Although this section uses the BSD file system[4] as an example, the problems are present in most commercial file systems in use today. The major reason that existing file systems will not scale with technology is that they perform too many random and synchronous disk operations.

A disk storage manager design is strongly influenced by the expected workload it must support. LFS is designed to support the workload of the office/engineering environment. This environment has been characterized (see[5]) by a large number of relatively small files (less than 8 kilobytes) whose contents are accessed sequentially and in their entirety. The average file life time is short, less than a day before it is overwritten or deleted. Efficient handling of file systems containing both small, rapidly changing files and large files is needed by this and many other UNIX environments.

File creation example

The performance problems of the BSD file system in the office/engineering environment can be seen by examining the disk access patterns caused by small file creation. Figure 1 shows the disk accesses required to create two single-block files in different directories. The UNIX system calls used to create the files were:

```
fd = creat("dir1/file1", 0);
write(fd,buffer,blockSize);
close(fd);
fd = creat("dir2/file2", 0);
write(fd,buffer,blockSize);
close(fd);
```

Figure 1 illustrates the disturbing access patterns of many small random and synchronous writes.
Each *creat* system call causes a random synchronous write of both the allocated inode block and the directory block. Synchronous writes of directory and inode modifications are intended to limit the damage caused by system crashes. The file data blocks and inode blocks for the directories are allocated in a manner that allows fast reads. Unfortunately, when files are small, allocations cause small random writes. The file data blocks and inode blocks for the directories are written asynchronously but still randomly. The total disk I/O in this example includes 8 random writes of which half are synchronous. Because of these random accesses, only a small fraction of the maximum disk write bandwidth can be used by the file system. The synchronous updates effectively limit the speed of file creation and deletion to the disk's speeds. For example, a .9-MIPS DEC MicroVaxII using the BSD file system can create and delete an empty file in 100 milliseconds. A 14-MIPS DEC DecStation 3100 using the same file system can create and delete an empty file in 80 milliseconds. Because of the synchronous disk I/O, an order-of-magnitude increase in CPU speeds causes only a 20 percent increase in program speed!

LFS storage manager design

Efficiently supporting office/engineering workloads means eliminating small synchronous writes. This section presents the major data structures and algorithms used by the LFS storage manager to accomplish this goal. The algorithms include those for file writing, file reading, and disk free space management.

File Writing

The key idea of LFS is to make writes fast by accessing the disk sequentially and asynchronously. LFS collects the changes to the file system in the file cache, packs them together, and writes them to disk in large sequential disk transfers. Modified file data blocks, directory blocks, and file system metadata such as inode blocks, are packed together and written sequentially to disk. LFS provides the abstraction that data is added to the file system in an append-only log format. The log format implies that LFS never updates disk blocks in place. This feature is the key to LFS's fast recovery and its ability to eliminate synchronous writes.

Because all writes are asynchronous, LFS uses the file cache as a write buffer that accumulates changes to the file system and performs speed matching between the CPU and disk subsystem. Bursts of small writes to the file system are combined together in the file cache and converted into large transfers. This conversion means that, unlike

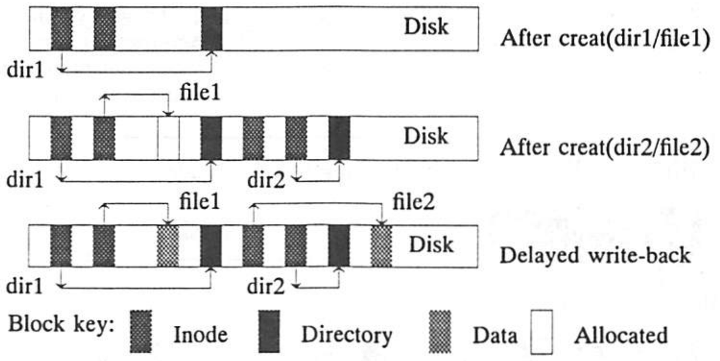


Figure 1: BSD file system file creation

Disk image of the BSD file system at different points in the creation of two files. The state is shown after each *creat* system call and after the delayed write-back has occurred. Each *creat* call forces the modified directory block and newly allocated inode to disk. For example during the first *creat*, file1's inode and dir1's inode and directory data block are written to disk. The write system calls allocate blocks on disk but the blocks are not written to disk until the delayed write-back occurs. After the second *creat*, the data block of file1 is allocated but is not written to disk until the delayed write-back.

the UNIX file system, file creation and deletion speeds in LFS are tied to the disk's maximum bandwidth and not its latency. Figure 2 shows the single disk access generated by the LFS storage manager executing the file creation example. In LFS, the *creat* system call simply allocates the file's number and modifies the directory and inode in memory. No synchronous disk writes are performed. The modified file and directory blocks are packed together and written in a single transfer. This form of I/O can be an order-of-magnitude faster than the many random write transfers done by the UNIX file system. For workloads that are limited by disk write performance, LFS permits files of any size to be written to disk at near maximum bandwidth. The elimination of synchronous writes allows LFS to scale with CPU speeds.

File reading

This section describes the data structures and algorithms used to read files from an LFS file system. Given the very large file caches that future systems will have, it is likely that read performance will have only a small impact on overall system performance. Nevertheless, LFS's read performance will match or exceed the performance of current read-optimized file systems in many cases. Although LFS always writes blocks sequentially, it still maintains the same data structures that permit fast read access in the UNIX file system.

Inode map

The major difference between file location in LFS and UNIX is that in LFS, inodes are no longer at fixed locations on disks. The append-only log abstraction provided by LFS requires that inodes be written to a new location on disk every time they are modified. LFS quickly locates inodes using a data structure called the **inode map**. The data structure maintains a mapping between an inode number and the current disk address of the inode. The inode map also keeps the inode status (allocated or free), the file's access time[1], and a version number that is updated every time the file is truncated to length zero. The version number usage is discussed in the 'Free space management' section.

For file systems having large numbers of files the inode map can get large. To reduce memory usage, LFS partitions the map into blocks that are cached like regular files. It is expected that the blocks mapping active files will stay memory resident and impose little overhead to the file inode fetch. Modified inode map blocks are written to the log like any other file block. The timing of inode map block writes and the recovery of the inode map after crashes are discussed in the section named 'Crash recovery'.

Except for the address lookup using the inode map, the file reading algorithm of LFS is identical to that of existing UNIX file systems. The format of inodes and indirect blocks is unchanged. For files that are written in their entirety, the log layout algorithm places the data blocks sequentially on disk. The read performance of such a file is excellent because the inode and all of the file's data blocks are located close together on disk.

Free space management

An LFS file system must manage the disk free space to keep open large regions of consecutive disk sectors. LFS simplifies this task by dividing the disk storage into large fixed-size pieces called **segments**. The idea is that the sequential log abstraction of LFS need not be totally sequential on disk. What really matters is that the log is written in large enough pieces to support I/O at near-maximum disk

[1] The file's access time is kept in the inode map because it is the only file attribute that is updated when the file is read. Keeping the access time in the inode map rather than the inode allows faithful implementation of the UNIX file access time semantics without inodes constantly moving every time a file is read.

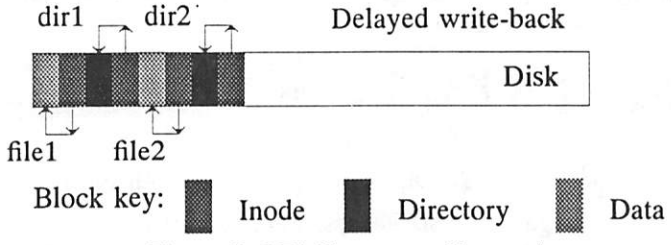


Figure 2: LFS file system file creation

Disk image of the LFS file system after two files are created. When a disk write is started, all modified file system blocks are packed together and written to disk in a single transfer. Note that the formats of directories and inodes are the same as in the BSD example (Figure 1). Rather than placing the blocks to optimize reading, LFS writes all changes sequentially to disk. LFS performs the 8 writes in one large transfer. Unlike the BSD example, all writes are sequential and none are synchronous.

bandwidth. This can be achieved by sizing segments so that the disk seek at the start of a segment write is amortized across a long data transfer time. The test presented in the last section used a segment size of one megabyte.

Segment summary blocks

Each LFS segment contains a few sectors of summary information that identify the contents of the segment and allow segments to be formed into a linked list. The information is kept in a region of the segment called the **summary block.** The append-only log abstraction can be visualized by following the links between segments.

For each block in the segment, the summary blocks indicates the file number of the block's file and the position of the block within the file. All the information needed for the summary blocks of a segment is available when a segment is formed in memory. The cost of the summary blocks is small in terms of CPU, disk space, and transfer time overheads.

Segment cleaning

The segmented-log structure of LFS reduces the free space management problem to that of finding a free segment to write. During normal operations segments will become fragmented: blocks in a segment will be overwritten or deleted, leaving the segment partially utilized. LFS generates free segments, called **clean segments**, from fragmented segments using an operation called **segment cleaning.** During segment cleaning two or more fragmented segments are read into memory, combined, and appended to the log on disk. Segment cleaning in LFS is simply a form of incremental **garbage collection**[6] where the fragmented segments are compressed together to create space to write new segments. LFS implements cleaning by reading the live blocks of a segment into the file cache and then using the cache write-back code to combine and copy the blocks into a new segment. The cleaning algorithm proceeds in two phases. During the first phase the live blocks of fragmented segments are identified and read into the cache. The second phase combines the blocks into a new segment and writes them to disk. Segment cleaning in LFS can be overlapped with normal file system operations. Files can be read and written while segments are being cleaned.

Block allocation determination

In order to perform segment cleaning efficiently, the LFS storage manager must be able to identify the "owner" (file number and block offset) of each block in a segment. It also needs to determine if a block is **live**, belonging to an active file, or if a block is **dead**, having been overwritten or truncated. LFS identifies blocks using the segment summary blocks, the inode map, and inodes. The block identification algorithm works for files and directories as follows:

1) The segment's summary block is used to determine the file number and block offset of the block being cleaned. Included in the summary entry is the file's version number from the inode map when the block was written. If the version number does not match the current version number of the file, the block is known to have been deleted or overwritten.
2) If the previous step fails to determine the allocation status of the block, the inode and any indirect blocks that map the block of the file are examined to see if the block is still part of the file. The block is classified as being live if it is still a member of the file.

Using these steps the cleaning algorithm can tell which blocks need to be copied into the new segment. Since total overwrite or deletion are the most common write access modes to files in the workstation environment, Step 1 is able to determine the live blocks quickly. Even if Step 2 must fetch the inode and indirect blocks to check the block's status, these blocks are exactly the blocks that must be fetched and modified anyway when the block of interest is copied to a new segment.

Choosing segments to clean

Although cleaning full segments will not harm the system, it is desirable to choose the segments with the most free space. To keep this information available, LFS keeps a data structure called the **segment usage array** that keeps an estimate of the number of live blocks in each segment in the system. The array is updated when files are truncated or overwritten and when segments are written or cleaned. Since segments are large and the usage array only takes a few bytes per segment, the array is small enough to stay memory-resident. Since the usage level of nonclean segments is used only as a hint during cleaning, costly exact crash recovery of this data structure is not needed.

Cleaning is activated either when the number of clean segments drops below a threshold value or when a user-level process initiates it. The user-level process interface allows cleaning to be initialized at night or other times of slack usage. Segments are cleaned until all segments are either clean or contain at least a file-system-settable fraction of live blocks.

Segment write timing

Because LFS does not perform synchronous writes, algorithms are needed to decide when segments should be written to disk. Segment writes in LFS are initiated by conditions similar to those that cause UNIX file data blocks to be written. A segment write is generated triggered by one of three conditions:

Cache full

The file cache may request a segment write when it detects a shortage of clean blocks in the cache. This condition will be triggered when many changes are made to the file system and the file cache is filled with dirty blocks.

Cache write-back

The file cache may request a segment write to start if it detects modified blocks older than a certain age threshold. The age threshold is much like the delayed write-back policy of UNIX. The current LFS uses a threshold of 30 seconds.

Sync request

A program executing a **sync** or **fsync** system call can cause data to be pushed to disk. The system call will block until the dirty blocks are written to disk.

Note that either of the last two conditions can cause a partial segment to be written because the file cache may not contain enough data to fill an entire segment. This may appear wasteful but since there were not enough dirty blocks for the entire segment this is a situation where the system was running much below capacity. Partially-written segments are handled by the segment cleaning algorithm like any other fragmented segment.

Crash recovery

The last part of the LFS design presented in this paper is the fast crash recovery system. System crashes can cause file systems to become inconsistent when disk operations are terminated by the crash or delayed writes are not completed. The log structure of LFS allows techniques commonly used in data base systems[7] to be used by LFS for fast recovery from system crashes. Unlike the UNIX file system, which must scan the entire disk after a crash to repair damage, LFS need only examine the tail of the log to find crash damage. LFS speeds recovery even more by periodically marking places in the log where the file system is known to be consistent. From these consistent points, called **checkpoints**, LFS can attach the file system without fear of inconsistent metadata.

The current implementation of LFS does not fully implement the high performance, reliable crash recovery mechanism designed for LFS. A much simpler algorithm with zero recovery time is used in the current implementation and described in this section. The current implementation has more overhead during normal operations and is more vulnerable to data loss at crashes than the recovery system LFS will ultimately use.

Checkpoints

LFS provides for the saving of dynamic file system state using an operation called a **checkpoint**. During a checkpoint, all of the memory-resident data structures that describe the current state of the file system are written to a known disk location called the **checkpoint region**. The checkpoint marks a consistent state of the file system. To allow for recovery from crashes during checkpoints, two checkpoint regions are used and checkpoint writes alternate between them. The checkpoint includes a timestamp that can be used to determine which region is the most recent checkpoint.

During a checkpoint, all modified blocks in the cache including all file system metadata blocks such as the inode map and segment usage array, are packed together and written to segments. Once all modifications are safely on disk, a checkpoint region is written that contains a pointer to the last segment written and the locations of the inode map and segment usage map. Crash recovery consists of nothing more than the normal file system "mount" code that uses the last checkpoint area to recover the file system state. Unfortunately, if the system crashes without writing the cache to disk, any changes made to the file system since the last checkpoint will be lost. The window of vulnerability can be controlled be setting the checkpointing interval. For example, our current checkpointing interval of 30 seconds means that in the worst case, changes made in the thirty seconds before a crash may be lost.

Ultimately LFS will be able to recover the information written between the last checkpoint and the crash. Using information in the segment summary blocks, LFS can "roll forward" from the last checkpoint, updating metadata structures such as the inode map and indirect blocks. This system will not require the entire file cache to be written on every checkpoint. Recovery will be slightly slower because the portion of the log between the last checkpoint and and the crash must be processed.

Related Work

LFS borrows ideas from many other systems. This section lists some of these systems. Organizing a file system as an append-only log is not a new idea. Several other file systems, motivated by the advent of write-once media such as optical disks, have used similar mechanisms. Write-once storage managers with random access to files include SWALLOW[8], the Optical File Cabinet[9], and others[10]. These file systems were intended principally for archival storage and not as high-performance file servers. Another way of viewing the LFS design is to see its roots in file systems like Cedar[11] that use logging to improve write performance and recovery time. The difference between LFS and other logging systems is that a read-optimized copy of the data is not kept and hence a copy does not need to be updated. The read-optimized format is not needed because reads are infrequent and the log's format is already well optimized for most file reads. The writing of several files to disk in one operation is much like the concept of **group commit**[12] found in database literature. Using group commit, database systems such as IBM's FASTPATH[13] delay writing so that several transactions can be committed in a single I/O.

An interesting related area of research is main-memory database systems. The metrics used to evaluate these systems are checkpoint overhead (flushing the dirty blocks to disk) and crash recovery speed. These metrics are similar to the design goals of write-optimized file systems. Main-memory database designs use logging and large asynchronous writes[1214].

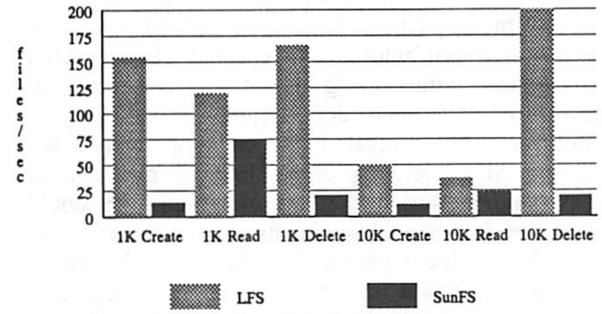


Figure 3: Small file I/O test
Measurements of creating, reading, and deleting many 1K and 10K files using LFS and the SunOS file system. The creation phase of the test measured the speed at which 10000 one-kilobyte and 1000 ten-kilobyte files could be created. Following the creation, the file cache was flushed and all the files were read (in the same order as they were created). Finally, we measured the speed at which the files could be deleted. All performance measurements are presented in number of files created, read, or deleted per second.

Performance of LFS

This section presents the performance of the LFS implementation running under the Sprite operating system. For comparison purposes, the same tests were also run under SunOS 4.0.3 using Sun's version of the BSD fast file system. The benchmarks were designed to demonstrate the important features of LFS; they were not meant to provide a thorough comparison between LFS and the SunOS file system.

The machine used for the test was a Sun-4/260 (16.6Mhz SPARC CPU) equipped with 32 megabytes of memory, a Sun SCSI3 HBA, and an WREN IV disk (1.3 MBytes/sec maximum transfer bandwidth, 17.5 milliseconds average seek time). For both LFS and SunOS, the disk was formatted with a file system having around 300 megabytes of usable storage. An eight-kilobyte block size was used by SunOS while LFS used a four-kilobyte block size and a one-megabyte segment size. The system was running multiuser but was otherwise quiescent during the test.

Small-file I/O

The first test measures the creation, reading, and deletion speeds of small files. The test consisted of creating 10 megabytes of small files, followed by flushing the file cache and reading all the files from disk. After reading all the files, they were deleted. Figure 3 presents results of test runs with files of size one kilobyte and ten kilobytes. The results are normalized to files per second created, read, and deleted.

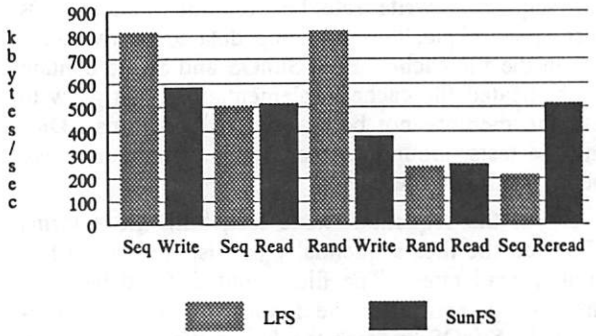


Figure 4: Large file I/O test
Transfer rates for reading and writing an 100 megabytes file. The figure shows the rate in kilobytes per second to create and write 100 megabytes sequentially, read 100 megabytes sequentially, write 100 megabytes randomly to the file, and read 100 megabytes randomly from the file. The final test was to read 100 megabytes sequentially after randomly writing the file.

The asynchronous file creation and deletion of LFS permits substantial gains in performance over the synchronous SunOS file system. SunOS performs small synchronous disk I/Os for each creation and deletion while LFS packs many changes into segments and pushs them to disk in megabyte transfers. This accounts for much of LFS's order-of-magnitude faster small file creation and deletion. While the SunOS performance is related to the disk access time, the LFS is CPU-bound on these tests. LFS performance will scale with CPU speeds until it is ultimately limited by the disk sequential-write bandwidth. Because the files are packed tightly together in segments, the read performance of LFS is excellent. Update-in-place file systems such as SunOS must spread files out to support file growth, thus hurting the read performance during these tests.

Large-file I/O

The second test measures the performance of writing and reading a 100-megabyte file from a newly created file system. The test consisted of five stages: writing a 100-megabyte file sequentially, reading the file sequentially, writing 100 megabytes randomly to the file, reading 100 megabytes randomly from the file, and rereading the file sequentially again. The test program used an eight-kilobyte request size. Figure 4 presents the transfer rates in kilobytes/second for each test.

As expected, LFS's write performance comes close to the maximum write bandwidth of the disk. Unlike the SunOS file system, LFS's write bandwidth is independent of how the file is written. LFS especially excels on the random write test because the random file writes become sequential writes when packed into segments. The SunOS file system must do random I/O to update the file in place. The random write rate for LFS is greater than the sequential write rate because the random I/Os were not unique, thus allowing data to be overwritten in the file cache. Both SunOS and Sprite contain sophisticated file cache implementations that grow to use the memory not being used by programs. During the tests, around 15 megabytes of memory were being used as a file cache.

For the sequential write test, both file systems allocated the files sequentially on disk and hence had similar read rates. The file layout differed between the file systems after the random writes were performed. SunOS updates the file in place while LFS writes the file to disk in the order the blocks are written. When the file is randomly read, both file systems must do random disk I/O to process the requests and thus they have equivalent read performance. The read performance of the file systems differed when the file was read sequentially after the random updates. For SunOS, the blocks were sequential on disk while LFS had to do random disk reads to fetch the blocks. This example demonstrates file access patterns in which conventional update-in-place file systems will have better disk read performance than LFS. LFS's sequential read performance can be reduced if the read access pattern is different than the write access pattern. An example for this would be sequentially reading a randomly written file. Note that the performance reduction only occurs in reads that miss in the file cache and would be satisfied sequentially by a conventional file system and not by LFS. Randomly reading a large sequentially written file will cause random I/O in both file systems.

Cleaning cost

During the tests described above, no segment cleaning was done in LFS. File I/O has traditionally been very bursty so we hope that much of the cleaning can be done using the idle cycles of the disk subsystem. This section examines the cost of segment cleaning and its effect on sustained performance.

The cost of segment cleaning is directly related to the utilization, or number of live blocks, in the segments being cleaned. Segments with no live blocks have no cost while full segments are expensive to clean and yield almost no free space. To evaluate the impact of cleaning on LFS performance, the rate at which bytes can be cleaned was measured for segments with varying utilization. The results of these tests are displayed in Figure 5. The varying utilization was generated by creating many one-kilobyte files, deleting some fixed percentage of them, and measuring the rate at which LFS could clean the fragmented segments. As expected, the more utilized the segment was, the longer it took to clean and the lower the rate at which clean segments could be generated. It is clear that cleaning highly utilized segments will significantly reduce the write-throughput of LFS.

Note that the x-axis of Figure 5 is the average utilization of the segments at cleaning time and not the overall disk-space utilization. For example, using 80% of the disk does not imply that the system will have to clean segments with 80% utilization. For non-synthetic workloads, segment utilization will form a distribution having a mean equal to the overall disk utilization. The shape and variance of the distribution are controlled by many factors including; file system access patterns such as amount of locality in file updates, LFS segment layout algorithms, and segment cleaning algorithms. It is currently not known what the segment distribution looks like for nonsynthetic workloads. The test presented in this section measured a worst case

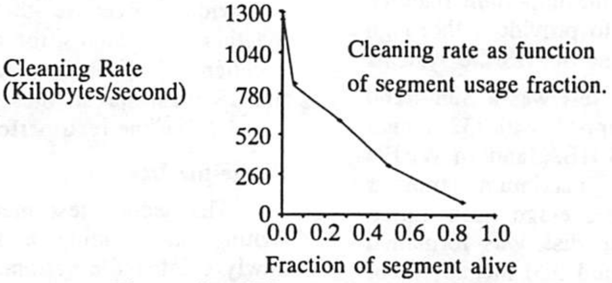


Figure 5: LFS segment cleaning rate

Measured rate of segment cleaning as a function of segment utilization. The rate is presented in kilobytes per second that clean segments can be generated.

scenario for LFS. Under non-synthetic workloads it would be highly unlikely that all of the segments would have the same fragmentation. The write performance of LFS will be controlled by how well LFS can hide cleaning cost during idle cycles. If cleaning can not be hidden, the question will be how full LFS can allow the disk to become and still keep the cleaning cost down.

Conclusion

We have presented a file system that attacks the I/O bottleneck problem by accessing the disk asynchronously and sequentially. The file system is structured as append-only segmented log for high write performance and fast crash recovery. File system metadata structures are maintained to allow high read performance. A segment cleaning operation is supported keep segments available for writing and to support high average disk space utilization.

Although LFS has been implemented and is doing very well on micro-benchmarks, the real test of a file system is its performance over months and years of use. As of this writing LFS has not been subjected to a "real" workload for extended periods of time. It is from these workloads that the overheads due to cleaning can be evaluated. The immediate plans for LFS include placing it in continuous use by the Sprite user community and examining its performance.

Acknowledgments

The work described here was supported in part by the National Science Foundation under grant CCR-8900029, and in part by the National Aeronautics and Space Administration and the Defense Advanced Research Projects Agency under contract NAG2-591. Diane Greene, Mary Baker, and John Hartman provided helpful comments on drafts of this paper.

References

1. John K. Ousterhout and Fred Douglis, "Beating the I/O Bottleneck: A Case for Log-structured File Systems," UCB/CSD 88/467, Computer Science Division (EECS), University of California, Berkeley, Berkeley, CA (October 1988).
2. John K. Ousterhout , Andrew R. Cherenson, Frederick Douglis, Michael N. Nelson, and Brent B. Welch, "The Sprite Network Operating System," *IEEE Computer* **21**(2) pp. 23-36 (1988).
3. David A. Patterson , Garth Gibson , and Randy H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," *ACM SIGMOD 88*, pp. 109-116 (Jun 1988).
4. Marshall K. McKusick , "A Fast File System for Unix," *Transactions on Computer Systems* **2**(3) pp. 181-197 (1984).
5. John K. Ousterhout, Herve Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson, "A Trace-Driven Analysis of the Unix 4.2 BSD File System," *Proceedings of the 10th Symposium on Operating System Principles*, pp. 15-24 ACM, (1985).
6. D. Ungar, "Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm," *Proceedings of the Software Engineering Symposium on Practical Software Development Environments*, pp. 157-167 (Apr 1984).
7. Jim Gray, "Notes on Data Base Operating Systems," pp. Springer-Verlag in *Operating Systems, An Advanced Course*, (1979).
8. D. Reed and Liba Svobodova, "SWALLOW: A Distributed Data Storage System for a Local Network," *Local Networks for Computer Communications*, pp. 355-373 North-Holland, (1981).
9. Jason Gait, "The Optical File Cabinet: A Random Access File System for Write-Once Optical Disks," *IEEE Computer* **21**(6) pp. 11-22 (1988).
10. Ross S. Finlayson and David R. Cheriton, "Log Files: An Extended File Service Exploiting Write-Once Storage," *Proceedings of the Eleventh Symposium on Operating System Principles*, pp. 129-148 (November 1987).
11. Robert B. Hagmann, "Reimplementing the Cedar File System Using Logging and Group Commit," *Proceedings of the 11th Symposium on Operating Systems Principles*, pp. 155-162 (November 1987).
12. David J. DeWitt, Randy H. Katz, Frank Olken, L. D. Shapiro, Mike R. Stonebraker, and David Wood, "Implementation Techniques for Main Memory Database Systems," *Proceedings of SIGMOD 1984*, pp. 1-8 (June 1984).
13. IBM, "IBM IMS Version 1 Release 1.5 Fast Path Feature Description and Design Guide," G320-5775, IBM World Trade Systems Centers (1979).
14. Robert B. Hagmann, "A Crash Recovery Scheme for a Memory-Resident Database System," *IEEE Transaction on Computers* **C-35**(9)(September 1986).

Mendel Rosenblum is a graduate student working towards his Ph.D. in Computer Sciences at the University of California, Berkeley. He is currently working as a research assistant on the Sprite network operating system project. His interests include operating systems, programming languages, and computer architecture. His electronic address is mendel@sprite.berkeley.edu .

John K. Ousterhout is a Professor in the Department of Electrical Engineering and Computer Sciences at the University of California, Berkeley. His interests include operating systems, distributed systems, user interfaces, and computer-aided design. He is currently leading the development of Sprite, a network operating system for high-performance workstations. In the past, he and his students developed several widely-used programs for computer-aided design, including Magic, Caesar, and Crystal. Ousterhout is a recipient of the ACM Grace Murray Hopper Award, the National Science Foundation Presidential Young Investigator Award, the National Academy of Sciences Award for Initiatives in Research, the IEEE Browder J. Thompson Award, and the UCB Distinguished Teaching Award. He received a B.S. degree in Physics from Yale University in 1975 and a Ph.D. in Computer Science from Carnegie Mellon University in 1980. Reach John electronically at ouster@sprite.berkeley.edu .,

Reach the authors at University of California, Berkeley; Computer Science Division; Berkeley, CA 94720.

USENIX Summer Conference
June 11-15, 1990
Anaheim, California

David C. Steere, James J. Kistler, M. Satyanarayanan – School of Computer Science, Carnegie Mellon University

Efficient User-Level File Cache Management on the Sun Vnode Interface

ABSTRACT

In developing a distributed file system, there are several good reasons for implementing the client file cache manager as a user-level process. These include ease of implementation, increased portability, and minimal impact on kernel size. For reasons of compatibility it is also desirable to use a standard file intercept mechanism on the client. The Sun VFS/Vnode file system interface is such a standard. However, this interface is designed for kernel-based file systems, and a user-level cache manager that used the Vnode mechanism would pay a large performance penalty due to the high number of kernel to cache manager context switches per file system call.

This paper describes our solution to the problem for the Coda file system. By using a relatively small amount of kernel code to cache critical information, we are able to retain the much larger and more complex components of the Coda cache manager in a user level process. The measurements of Coda presented here confirm the performance benefits of this strategy, and indicate the relative merits of caching different kinds of information in the kernel.

1. Introduction

In this paper we describe and evaluate our approach for implementing a user-level client cache manager on top of the Sun Vnode interface. Our work was carried out in the context of the *Coda file system*[5], a descendant of the *Andrew file system* (AFS)[4]. Coda is a highly available file system for a large-scale distributed computing environment composed of Unix workstations. It provides resiliency to server and network failures through the use of two distinct but complementary mechanisms. One mechanism, *server replication*, stores copies of a file at multiple servers. The other mechanism, *disconnected operation*, is a mode of execution in which a caching site temporarily assumes the role of a replication site. Disconnected operation is particularly useful for supporting portable workstations.

The design of Coda optimizes for availability and performance, and strives to provide the highest degree of consistency attainable in the light of these objectives. In the presence of network partitions, Coda places availability above consistency and allows potentially conflicting updates to occur. Such updates are detected as *inconsistencies* at the earliest possible time. The most complex part of Coda is the client cache manager, known as *Venus*. Besides cache management, Venus is responsible for emulation of Unix semantics, coordination of the replica control algorithm, detection of inconsistencies, and disconnected operation.

Given the complexity of Venus, we saw good reasons for it to be a user-level process rather than part of the kernel. Our experience has been that kernel code takes much longer to develop, debug and maintain. We felt that designing Venus as a user-level process would substantially ease our implementation effort and also increase Coda's portability.

A different design consideration, with conflicting implications, was our desire to adhere to standards. We saw many advantages, particularly that of portability, in using the Sun Vnode interface[2]. This interface is a *de facto* standard for file system call interception in Unix. Unfortunately, it imposes a component-by-component interaction between the kernel and the cache manager when translating pathnames. Consequently, a naive user-level cache manager implementation on this interface would result in unacceptable overhead.

To reduce this overhead, we decided to build a caching module in the kernel, called the *MiniCache*. Since most Unix applications show strong locality of reference, we felt that this would achieve a large reduction in the number of calls to Venus. We also conjectured that much of this reduction could be achieved with a MiniCache whose size and complexity were a small fraction of that of Venus. Our conjecture has indeed proved to be true. The performance of Coda with the MiniCache is comparable to that of the current production version of AFS, whose cache manager is in the kernel.

2. Design Considerations

The dominant influence on the design of the MiniCache was the Vnode interface. This interface was created in conjunction with the Sun *Network File System* (NFS)[3]. It behaves as a file system multiplexor, permitting multiple file systems to coexist in the same name space. The interface specifies two types of operations: those on a *Virtual File System* (VFS), and those on a *Virtual Inodes* (Vnode). VFS operations, such as `mount` and `unmount`, pertain to the binding of different file systems into one name space. Vnode operations, such as `open`, `close`, and `getattr`, pertain to individual files and directories. To add a new file system, one need only provide a VFS *driver*, which provides an entry point for each VFS and Vnode operation.

An implicit assumption of the Vnode interface is that the VFS driver resides in the kernel. The most obvious manifestation of this assumption is the strategy used to perform pathname translation. Rather than passing the entire pathname to the driver, translation is performed on a component-by-component basis. Although this simplifies symbolic link traversal across file systems, it would impose a serious performance penalty on an out-of-kernel driver when translating multi-component pathnames.

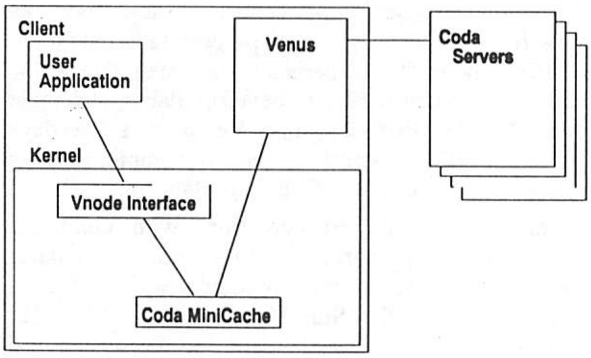


Figure 1: Coda Structure

We therefore decided to split the VFS driver into two pieces: a small piece inside the kernel (the MiniCache), and a more sophisticated piece outside the kernel (Venus). Essentially, the MiniCache behaves as buffering agent between the Vnode interface and Venus, as illustrated in figure 1. When an application program generates a file system call on an object in Coda, it is intercepted by the Vnode interface and routed to the MiniCache. If the MiniCache has sufficient information, it services the call. Otherwise, it passes the request to Venus. Venus handles the request, possibly contacting Coda servers in the process. Control is returned to the application via the MiniCache and the Vnode interface.

A consequence of the splitting of the VFS driver is that state is shared between its two components. The correctness of the system requires coherence in this shared state. It is clear to see that mutating file system calls will change MiniCache state, and that these changes must be propagated to Venus and thence to the servers. But there are also situations in Coda where Venus may need to invalidate MiniCache state. Such invalidations may arise in a number of different ways. First, a server may invalidate a cache entry in Venus because of a modification at another client. Second, authentication tokens held by Venus on behalf of a user may expire, implicitly revoking his privileges. Third, Venus may discover that an object is inconsistent, and therefore discard all local knowledge of it. Shared state also has an important implication for the communication mechanism between the MiniCache and Venus: both parties need to be able to initiate message exchanges.

A broader consideration is the need to strike the right balance between complexity and efficiency. The MiniCache needs to be sophisticated enough to improve performance substantially, but should not be so complex that it amounts to moving Venus completely into the kernel. Further, the MiniCache needs to be particularly efficient because it is in the critical path of many operations.

3. The MiniCache

This section describes four major aspects of the MiniCache. The first subsection discusses the main data structures. The second describes the communication between Venus and the MiniCache. The third presents the performance enhancements we have implemented, and the fourth discusses a limitation of the Vnode interface that we encountered when dealing with the `exec` system call.

3.1 Data Structures

The MiniCache consists of two interdependent caches. The first cache, the Cnode cache, contains a set of *Cnodes*, which are Vnodes augmented with Coda-specific fields. The latter include the *CFid* (an identifier unique to each Coda object) as well as fields used in the performance enhancements described in Section 3.3. For efficiency, Cnodes are kept on an internal freelist rather than being deallocated. To provide Venus with a fast access path, this cache can be looked up via a hash table keyed on CFids.

The second cache, the name cache, contains precomputed translations of pathnames. Each entry in the cache is indexed by a triple: a pointer to the Cnode for a directory, a component name in that directory, and a user id. A Coda object may have multiple entries in the cache, one for each user who has accessed the file recently. The value of the cache entry is a pointer to the Cnode corresponding to that component. The name cache supports the basic operations of insertion, lookup, and deletion. Deletion happens to be a particularly complex operation since it occurs in multiple flavors. The simplest case is when a deletion occurs via the Vnode interface. More complex cases occur when Venus deletes entries for one of the three reasons cited in Section 2.

The functioning of the name cache can be best understood by following a typical pathname translation. At any step in the translation one has a pointer, *DCP*, to the Cnode of a directory, a name, *N*, in that directory, and the id, *U*, of the user on whose behalf the translation is being performed. If the triple *(DCP, N, U)* is missing from the name cache, Venus is contacted to fill the entry. Translation continues by replacing *DCP* by the value of the cache entry, and N by the next component of the pathname, and repeating the cache lookup. Translation begins with *DCP* set to the root (for absolute pathnames) or the current directory (for relative pathnames).

Our name cache is similar in many ways to the Directory Name Lookup Cache (DNLC), supplied with the Sun Vnode interface[3]. Unfortunately the DNLC does not provide the full range of invalidation functions needed by Venus. Rather than modify it, we chose to build our own name cache.

3.2 Communication

In addition to the need for either peer to initiate action, the communication mechanism between the MiniCache and Venus needs to satisfy two conditions. First, it has to be interruptible in a manner that allows cleanup of state at both ends. Second, it must provide at-most-once semantics since many of the operations are not idempotent.

We initially tried using Sun RPC[6] for communication between the MiniCache and Venus. This would have enhanced the portability of the MiniCache, since Sun RPC is normally available in any kernel that has the Vnode interface. Unfortunately, we found it impossible to meet both the constraints mentioned above using that approach.

Consequently, we settled on a pseudo-device as the basis of our communication. The device is used to pass messages between the MiniCache and Venus. At-most-once semantics is trivially obtained because there are not retransmissions or timeouts. Interruptibility is ensured by careful implementation of the device driver. In addition, the pseudo-device supports concurrency by permitting an arbitrary number of outstanding messages.

Although the use of a pseudo-device limits portability to Unix-like systems, the communication code is limited to a small section of the MiniCache. The pseudo-device could easily by replaced by any other reasonable IPC mechanism.

3.3 Performance Enhancements

There are many conceivable ways in which the MiniCache can be used to improve performance. Given our desire to keep the MiniCache simple, we have only chosen to make those performance optimizations that provide substantial benefit relative to implementation complexity. We elaborate on these optimizations in this section.

The first, and most obvious, optimization is the reduction in MiniCache-Venus traffic during pathname translation. This reduces the average cost of the `lookup` Vnode operation, and is provided by the name cache described in Section 3.1.

The second optimization eliminates the need for the MiniCache to contact Venus on individual read and write operations. Besides reducing the number of MiniCache-Venus interactions, this substantially reduces the amount of data copying done by Coda. On a successful `open` of a file, the MiniCache saves a pointer to the Vnode of the locally cached copy. It can then service `rdwr` Vnode operations on the file by redirecting them to the local file system. `Readdir` Vnode operations on directories are handled in a similar manner. The MiniCache notifies Venus of `close` operations, giving Venus the opportunity to write a file back to the server if necessary.

The next performance enhancement we implemented was to provide an attribute cache, allowing `getattr` operations to be serviced by the MiniCache without contacting Venus. Such operations occur frequently, typically due to the `stat` system call which is extensively used by Unix applications. Space for the cache is provided as an extension of each Cnode, with a flag indicating whether the attribute fields are valid. The attributes for an object are invalidated when Venus flushes the object from its cache. Attributes are also invalidated in two other cases: when a file that has been modified is closed, and when a child of a directory is modified. Access permissions are correctly enforced for `getattr` since a user needs appropriate access rights to lookup the Cnode.

The fourth enhancement takes advantage of the presence of the name cache to reduce the number of `access` calls to Venus. `Access` is a Vnode operation that is used to determine a user's permissions on an object. Checking access rights in Coda is substantially more complex than checking Unix mode bits because Coda uses an access list mechanism with group inheritance. A full implementation of `access` in the MiniCache would greatly add to its complexity. Fortunately, we have been able to trivially implement an important subset of the full access check. It turns out that a large percentage of the `access` operations occur during pathname translation, and simply determine if a user has permission to lookup entries in a directory. Since name cache entries include the user id as part of the key, we are able to infer lookup permission on a cache hit. In all other cases, the MiniCache forwards access operations to Venus.

The last performance enhancement is to speed up symbolic link traversal by caching the link value within the Cnode. Symbolic links are frequently encountered during pathname translation, particularly in Unix environments that support heterogeneous machines. The symbolic link cache does not violate access constraints because a user needs to have proper access to look up the Cnode of the directory containing a symbolic link. In Unix, lookup permission on a directory automatically implies permission to read symbolic links in that directory.

3.4 Program Execution

In the course of our implementation we ran into a serious limitation of the Vnode interface: there is no provision for a VFS driver to be notified of exec and exit system calls. All the driver sees are read operations on the individual pages of an executable file. This is a problem for any stateful file system, since it now has to infer when to allocate and free internal state. For example, it needs to guess when a file must be cached, and when it is safe to flush it.

This problem is magnified for Coda since Venus cannot directly access kernel data structures. Although it is possible for the MiniCache to infer opens, it does not possess enough of Venus' state to determine when it is appropriate to flush a file. On the other hand, Venus knows when it is appropriate to flush a file, but does not possess enough kernel context to know if this would be safe. So the flushing or replacement of a cache entry by Venus is preceded in our implementation by a call to the MiniCache to determine if the operation is safe. Besides degrading performance, this approach also limits portability since it depends on specific details of the virtual memory subsystem.

4. Performance Evaluation

Our evaluation of the MiniCache focuses on two important questions. First, how much improvement does the MiniCache provide, both in absolute terms and relative to the current production version of AFS with an in-kernel VFS driver? Second, what is the individual contribution of each performance enhancement?

The primary basis for this evaluation is the Andrew Benchmark[1], consisting of a series of Unix operations on a subtree containing the source code of a Unix application. The input to the benchmark consists of 70 files, totalling about 200 Kbytes. The benchmark consists of five phases: MakeDir, which constructs a copy of the source subtree; Copy, which copies the source files to the new subtree; ScanDir, which examines the status of each file in the target subtree; ReadAll which scans every byte in every file in the target subtree; and Make, which compiles and links all the files.

Since the Andrew benchmark does not make extensive use of symbolic links, we use a different criteria for measuring the effect of caching them. Details of this test are provided in Section 4.3.

The tests reported here were conducted on a single client and server located on a single segment of Ethernet. Both machines were IBM APC-RTs with 12 MB of memory, running the Mach 2.5 operating system. To limit interference, nonessential processes on the client were killed before running the benchmarks. Although experimental control on the servers and the network was less strict, the low variance in our results indicates that this was not a problem.

4.1 Comparison with AFS

In estimating the absolute and relative performance benefits of the MiniCache we have chosen to focus on two phases of the Andrew benchmark. These phases, ScanDir and ReadAll, are the most demanding phases in terms of pathname translation. They involve no server interactions in Coda and AFS because they operate on data cached in an earlier phase of the benchmark. Hence the performance degradation due to an out-of-kernel VFS driver would be most apparent in these phases.

Table 1 presents the elapsed times, in seconds, of these phases averaged over four runs of the benchmark. With the MiniCache turned on, Coda performs as well as AFS. Our original goal of obtaining good performance without moving Venus into the kernel has thus been met. Without the MiniCache, substantial degradation is apparent. This is to be expected, since all Vnode operations are simply redirected to Venus, suffering at least two context switches in the process.

Configuration	ScanDir	ReadAll
Coda without MiniCache	44 (1)	71 (1)
Coda with MiniCache	26 (1)	43 (1)
AFS (in-kernel cache manager)	26 (0)	43 (1)

Running times in seconds of the third and fourth phases of the Andrew Benchmark. Numbers in parentheses indicate standard deviations.

Table 1: Performance Improvements Due to the MiniCache

4.2 Individual Contributions

In this section we present the observed performance improvement due to each of the enhancements described in Section 3.3. The metric used is the number of Vnode operations seen by Venus during the running of the Andrew benchmark. This is a completely deterministic metric, for three reasons: our measurements were made after an initial run to warm cache entries for the source subtree, the benchmark is small enough to avoid cache overflow, and target subtrees were deleted between benchmark runs.

Table 2 shows the reduction in number of Vnode operations due to the individual performance enhancements, with the listed operations accounting for about 80% of the total number in the benchmark. The table indicates that the name cache eliminates all `lookup` operations on Coda.

Although `rdwr` calls to Venus have been eliminated, our measurements indicate that there is not a correspondingly large savings in elapsed time. We suspect that this is because the cost of accessing the local disk far outweighs the cost of contacting Venus.

The number of `getattrs` that reach Venus has been reduced by nearly a factor of 8. The reason that `getattrs` are not totally eliminated is that attributes are flushed whenever a file is modified or a directory is updated. Thus the 83 getattrs in the `ScanDir` phase correspond to the 70 files that were created in the `Copy` phase and the 13 directories in which files were modified.

Table 2 also shows that our simple strategy for reducing `access` calls to Venus works very well: the number of such calls is reduced by nearly half. Complete elimination of `access` calls to Venus would have required much greater effort.

4.3 Symbolic Link Caching

To evaluate the effect of caching symbolic links, we wrote a small program to repeatedly invoke the `stat` system call on the head of a chain of symbolic links. The first name in the chain pointed to the second, which pointed to the third, and so on. We tried two variants of this test: one in which the symbolic links were relative, and the other in which the symbolic links were absolute pathnames that were 5 components long. The target file was the same in both cases, and all links and the target file were in the same directory.

Each `stat` generates a `lookup` on the name of symbolic link, which generates a `readlink` to get its value. This is then followed by a `lookup` on the name specified in the link and then a `getattr`. With relative pathnames, this amounts to two

VFS Operation	Enhancement		MkDir	Copy	ScanDir	ReadAll	Make	Total
`lookup`	Name Caching	off	33	745	1586	2139	886	5369
		on	0	0	0	0	0	0
`rdwr/readdir`	RdWr Intercept	off	0	332	136	498	2107	3073
		on	0	0	0	0	0	0
`getattr`	Attribute Caching	off	0	70	361	360	84	875
		on	0	0	83	0	29	112
`access`	Access Caching	off	0	70	241	500	202	1013
		on	0	70	68	234	201	573

This table contains a summary of the number of operations seen by Venus during a run of the Andrew Benchmark. The values for lookup are the number of lookup operations performed by Venus that succeeded. The Name Cache row indicates that all lookups of Coda files were handled in the kernel. ReadWrite is a combination of the number of `rdwr` and readdir Vnode operations seen by Venus.

Table 2: Reduction of Operations due to MiniCache

Number	Without Cache				With Cache			
of links	Rel Path		Abs Path		Rel Path		Abs Path	
1	84.0	(.21)	94.6	(.30)	9.9	(.07)	20.1	(.13)
2	160.4	(.19)	182.5	(1.56)	13.1	(.15)	33.2	(.10)
3	237.4	(.32)	268.3	(.29)	16.3	(.08)	46.5	(.12)
4	313.5	(.45)	359.1	(1.55)	19.4	(.14)	59.7	(.11)
5	393.2	(2.99)	443.9	(1.58)	22.6	(.08)	73.0	(.15)

This table contains the time to execute 10000 stat calls on symbolic links. Two kinds of links were tested, one using absolute pathnames as the values of the links and the other using relative pathnames. The number of links refers to the number of symbolic link expansions needed to find the real file. Each number is the average of four runs, with the standard deviation in parentheses.

Table 3: Performance of the Symlink Cache

lookups, one readlink, and one getattr. The use of absolute pathnames adds five more lookup calls, one for each directory in the pathname.

Table 3 shows mean and standard deviation of the elapsed time in seconds to perform 10000 stats on each link of the chains. The table indicates that it takes about 10 seconds longer per link with absolute pathnames than with relative pathname links. With the name cache enabled, all the lookups can be satisfied in the kernel. Thus the cost of looking up an entry in the name cache is roughly two tenths of a millisecond.

The table also shows that each additional link adds about 3 seconds with the symlink cache enabled, and about 77 seconds otherwise. Each link causes an additional readlink and lookup operation. Since the name cache was enabled in both cases, almost all of the 77 seconds is due to calls to Venus.

Symbolic links are used quite heavily in our environment. In the Coda source subtree alone, there are 439 symbolic links out of roughly 10,000 files and directories. The benefits of symbolic link caching are therefore quite substantial.

5. Conclusion

Although early versions of AFS demonstrated that efficient user-level cache management is possible, they depended on a customized system call intercept mechanism. When AFS adopted the Vnode interface, Venus was moved into the kernel to avoid excessive performance degradation. Rather than follow this approach, we decided to investigate the possibility of retaining Venus as a user-level process.

Our approach has been to move relatively simple, yet critical, pieces of Venus functionality into the kernel. Our analysis of representative Unix file system activity indicates that five such pieces (pathname translation, data read/write, attribute read, access checking, symlink expansion) account for the bulk of the Vnode operations. A key aspect of our solution is that the code we have added to the kernel relatively small and simple. Our results indicate that we are able to match the performance of an in-kernel client cache manager, AFS, in the most demanding phases of the Andrew benchmark.

Two aspects of the MiniCache limit its portability slightly. One is our use of a Unix device driver, rather than Sun RPC, for communication. The other is our dependence on code specific to the Mach virtual memory implementation in order to correctly handle the execution of files. These limitations are unfortunate, since portability was one of the factors motivating our use of the Vnode interface.

The concerns which motivated this work are fundamental to a distributed system. On one hand, performance concerns lead one to design the system as part of the kernel. On the other, issues of portability and maintainability suggest that development should proceed at the user-level. The key result of this work is that an efficient portable user-level cache manager can be built on the Sun Vnode interface. In addition, it can be achieved via a small and simple piece of code in the kernel.

References

[1] Howard, J., Kazar, M., Menees, S., Nichols, D., Satyanarayanan, M., Sidebotham, R., and West, M. Scale and Performance in a Distributed File System. *ACM Trans. Comput. Syst. 6*, 1 (Feb. 1988).

[2] Kleiman, S. Vnodes: An Architecture for Multiple File System Types in {Sun UNIX}. In *Summer Usenix Conference Proceedings, Atlanta* (1986).

[3] Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., and Lyon, B. Design and Implementation of the Sun Network File System. In *Summer Usenix Conference Proceedings, Portland* (1985).

[4] Satyanarayanan, M., Howard, J., Nichols, D., Sidebotham, R., Spector, A., and West, M. The ITC Distributed File System: Principles and Design. In *Proceedings of the 10th ACM Symposium on Operating System Principles, Orcas Island* (Dec. 1985).

[5] Satyanarayanan, M., Kistler, J., Kumar, P., Okasaki, M., Siegel, E., and Steere, D. Coda: A Highly Available File System for a Distributed Workstation Environment. *IEEE Trans. Comput. 39*, 4 (Apr. 1990).

[6] Sun Microsystems, Inc. RPC: Remote Procedure Call Protocol specification version 2. Tech. Rep. RFC-1057, SRI Network Information Center, June 1988.

David C. Steere received the B.S. degree in computer science from Worcester Polytechnic Institute, Worcester, MA, in 1988. He is currently pursuing a Ph.D. in computer science at Carnegie Mellon University. His research interests include distributed systems, communications, performance modeling, and operating systems. Mr. Steere is a member of the Association for Computing Machinery, Tau Beta Pi, and Upsilon Pi Epsilon.

James J. Kistler received the B.S. degree in business administration from the University of California, Berkeley, in 1982, and the Diploma in Computer Science from the University of Cambridge, England, in 1985. He is currently working toward the Ph.D.

degree in Computer Science at Carnegie Mellon University. He has been a member of the Coda project since its inception in 1987. His research interests are in distributed and parallel computing. Mr. Kistler is a member of the IEEE Computer Society and the Association for Computing Machinery.

Mahadev Satyanarayanan is an Associate Professor of Computer Science at Carnegie Mellon University. His research addresses the general problem of sharing access to information in large-scale distributed systems. In his current work on the Coda File System, he is investigating techniques to provide high availability without significant loss of performance or usability. An important aspect of this work is providing distributed file access to portable computers that may operate disconnected for significant periods of time. Earlier, he was one of the principal architects and implementors of the Andrew File System, a location-transparent distributed file system that addressed issues of scale and security. His work on Scylla explored access to relational databases in a distributed workstation environment. His previous research included the design of the CMU-CFS file system, measurement and analysis of file usage data, and the modelling of storage systems.

Professor Satyanarayanan received the PhD in Computer Science from Carnegie Mellon in 1983, after a Bachelor's degree in Electrical Engineering and a Master's degree in Computer Science from the Indian Institute of Technology, Madras. He is a member of the ACM, IEEE, and Sigma Xi, and has been a consultant to industry and government. He was made a Presidential Young Investigator by the National Science Foundation in 1987.

USENIX Summer Conference
June 11-15, 1990
Anaheim, California

A Filesystem For Software Development

David Hendricks – Sun Microsystems, Inc.

ABSTRACT

Successful software development often requires the duplication of source hierarchies, either to make snapshots of releases or to allow multiple developers to work in parallel on common sets of sources. Copying files is expensive in terms of space, time, and the administrative burden of keeping all the copies up-to-date. The Translucent File Service (TFS) is a special-purpose filesystem, transparent to user programs, that removes the need to copy files.

The TFS is a Sun Operating System (SunOS) filesystem with copy-on-write semantics. The TFS allows users both to share a file hierarchy and to have a private hierarchy into which files from the shared hierarchy are copied as they are modified. Consequently, users are isolated from each other's changes, as files in the shared hierarchy are guaranteed not to change. Files are only copied when they are modified, conserving disk space. The TFS was built to support Sun's version configuration and management tool, the Network Software Environment (NSE). The TFS is a mature filesystem that has been made a standard part of SunOS version 4.1.

This paper describes the semantics that the TFS provides, and presents several applications of the TFS. The implementation of the TFS is described, along with possibilities for future development.

Introduction

The Problem: Managing Copies of Files

In the realm of software development and maintenance, there are two scenarios which cause copies of source code to be created. The first is that of a number of developers working in parallel on a set of sources. In a large software project, the sources will be large enough that allowing only one developer at a time to modify the sources will not be efficient. Instead, each developer will need a private copy of the source, or at least a copy of the part of the source that he/she is working on, to avoid interfering with other developers.

The second scenario is the need to go back to an older version of the source, perhaps to debug a bug in a previous release of the software, or to investigate the changes between a previous release and the present one. Doing this correctly requires that a complete copy of the sources be made every time a release is made. It may also be necessary to make a copy of the compilers and other tools used to build the sources at the time of a release.

There are two drawbacks to all of this file copying. For one, it is expensive in time and space to copy files, especially for a large set of sources. Another drawback is that confusion can result later if source files in one of the copies get out of date. This can happen easily, because it is difficult for people to manage the tedious task of keeping all copies of the source up-to-date.

A Solution: Viewpaths

What is needed to avoid this confusion and waste of space is some mechanism to share files. One way to avoid file copying is to modify an individual program, like *make*, to use a *viewpath* to find files. The *viewpath* specifies an ordered list of directories to look for a file in. The search for a file terminates as soon as the file is found in one of the directories in the search list. At least two variants of *make* support viewpaths: *build* [1] and *new-make* [2]. The viewpath mechanism allows a programmer to insert a private working directory in the viewpath ahead of the reference source directory, so that source files with local bug fixes can be put into the private directory and used by *make* to build a binary. A major problem with modifying *make* in this manner is that changes to the viewpath can make it hard to determine which sources were used to build a binary.

An alternative to modifying a few programs to use viewpaths is to provide operating system support for viewpaths, so that the mechanism can be used by all programs. A number of DEC operating systems had such a mechanism: TOPS-10 allowed a second directory to be searched for a file, and TOPS-20 and VMS provided *search lists*, which allowed an arbitrary number of directories to be searched for a file. Ronald Brender [3] described how the TOPS-20 directory search list could be used to share source files in a large compiler project involving a high degree of simultaneity. Korn's 3-D filesystem [4] is one implementation of viewpaths on Unix.

Another Solution: Versioned Files

Another alternative which allows file sharing is to implement *versioned*files where multiple versions of a file are accessible, and a specific version is specified either explicitly in the file name (e.g., **foo.c[1.2]** selects version 1.2 of file **foo.c**) or by a process-specific table that specifies the versions to be selected when files are read. With the process-specific table, one user doing development can specify that he wants to see the current version of the sources, while another user, chasing a bug in a released version of the product, can look at the sources as they existed at the time of the release. Apollo's DOMAIN Software Engineering Environment (DSEE) [5] implements versioned files. In DSEE, all versions of a file are stored in one file on disk, and the DSEE filesystem's *read* operation extracts the parts of the file that appear in the version being read. The 3-D filesystem also implements versioned files. In this filesystem, a versioned file is stored as a directory which contains individual files for each version of the file. The DSEE saves more disk space than the 3-D filesystem by storing all versions of a file in one disk file.

One problem with the versioned file approach is that there must be a mechanism for passing process-specific version numbers to the file *read* routines. DSEE was only implemented on the DOMAIN operating system, which allows this to be done. For versioned files to work in Unix, either the file-handling routines would have to be implemented in user-level libraries, or Unix would have to be extended with new system calls to control version mapping. (Both of these were done for the 3-D filesystem, in two implementations: the former for a user-level implementation and the latter for a kernel implementation.)

A Transparent Viewpath Solution: The TFS

The problem is that Unix provides no operating system support for a viewpath mechanism. How can one be added? One possibility is to link programs requiring the viewpath semantics against a special library which intercepts system calls and applies the viewpath semantics. There are a number of problems with this: 1) it is not easy to determine the complete list of programs that need to use the viewpath, 2) programs supplied by other vendors that should use the viewpath will need to be re-linked to use the library, and 3) one cannot prevent a program that was built without the viewpath library from running. Problems (1) and (2) could be solved by putting the viewpath mechanism into a shared library that is dynamically linked to binaries at runtime. However, it would still be possible to run statically-linked programs that would not use the shared library, so problem (3) wouldn't be solved.

What is needed is a viewpath mechanism that can be interposed somehow between a user process and the file system, in such a way that the user process doesn't need to be modified to use the viewpath. In fact, the user process shouldn't even need to know that a viewpath is being used. The Translucent File Service (TFS) is such a mechanism: it is a special-purpose filesystem that transparently provides the functionality of viewpaths.

This paper will describe the semantics of the TFS, then will explain how the TFS can be applied to 1) make snapshots of directories without copying them, and 2) allow developers to work in parallel on a set of sources without copying the sources. (For another discussion of these two problems and possible filesystem solutions to them, see Hume [6].) Finally, the implementation of the TFS will be discussed. This discussion will explain how it is possible to make the TFS transparent to user processes.

Semantics of the TFS

TFS Layers

A directory seen through the TFS appears to be a normal Unix directory but is actually a number of *layers*, where each layer is a physical directory. The layers are joined by *searchlinks*. Each layer has a searchlink which contains the directory name of the next layer and is stored in a file named **.tfs_info** that the TFS hides from the user. The first layer seen through the TFS is considered to be the *front* layer, and layers found by following searchlinks from this layer are considered to be behind the front layer.

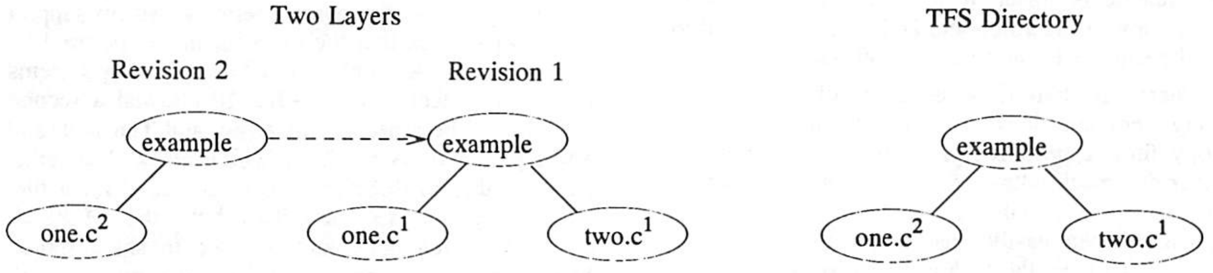


Figure 1: A 2-layer TFS directory

Figure 1 illustrates a 2-layer TFS directory named **example**, where "Revision 2" is the front layer and "Revision 1" is the back layer. The dotted line represents the searchlink from the Revision 2 **example** directory to the Revision 1 **example** directory. A layer, in this example, can be thought of as a revision of a directory; a snapshot of the directory's contents at some point in time. The newest revision of the directory is the front layer, and the rest of the layers are older revisions.

TFS Directory Contents

The file names seen in a TFS directory are the union of the file names seen in all the layers. If a file name exists in more than one layer, the file contents seen through the TFS will be the contents of the file in the front layer. For example, in Figure 1, the TFS directory **example** contains the two files **one.c** and **two.c**. The file **one.c** exists in both layers, so the file contents that the user sees will be those of the file in the front layer, or Revision 2. On the other hand, **two.c** doesn't exist in the front layer, so the the user sees its name and contents *showing through* from the back layer, Revision 1. Note that the front layer is *translucent*, or selectively transparent, because files show through from back layers unless files exist in the front layer to mask them. This translucency gives the "Translucent File Service" its name.

Copy-on-write

In the TFS, only the front layer can be written to; all other layers are read-only. Consequently, files are created and modified in the front layer. When a user modifies a file which is showing through from a read-only layer, the file is copied to the front layer before it is modified. This process is known as *copy-on-write.*

White-out

If a user wants to remove a file which is showing through from a read-only layer, the TFS cannot simply remove the file from the read-only layer, because by definition that layer cannot change. Instead, the TFS employs *white-out* to remove the file name from the TFS directory while leaving the file's contents intact in the read-only layer. For example, in Figure 1, if the user wanted to remove the file **two.c**, then the TFS would create a white-out entry for **two.c** in Revision 2. This white-out entry would mask the file that would otherwise show through from Revision 1. Note that if the user wanted to remove **one.c**, the TFS would both remove the file and create a white-out entry to prevent **one.c** from showing through from Revision 1. The white-out entries for a directory are stored in the same hidden **.tfs_info** file that contains the searchlink for the directory. There are two commands to manipulate white-out entries: *unwhiteout* removes a white-out entry, and *lsw* lists the white-out entries in a given directory.

Applications of the TFS

The copy-on-write property of the TFS allows the TFS to be used in ways that reduce the amount of file copying necessary to accomplish a software development task. Applications that want to make full use of the TFS need to build layers for the TFS to read by creating directories and setting searchlinks between them. (Applications set searchlinks by creating **.tfs_info** files and writing into each of them a searchlink, which is merely a pathname.) Once the layers have been built, the TFS filesystem is made visible with a *mount* system call. One application that builds TFS directory structures and relies on the TFS's functionality is the Network Software Environment (NSE), Sun's configuration management and version control tool [7].

Directory Revisions

One application of the TFS is as a mechanism to save multiple revisions of a directory. If one wanted to take "snapshots" of a directory as it changes through time, one could make a complete copy of the directory for each snapshot. This would consume a lot of disk space, as has been mentioned. The copy-on-write properties of the TFS can be used to minimize the amount of disk space needed to store the snapshots of the directory, as illustrated in Figure 2. Suppose at time 0 a TFS directory contains two layers: an empty front layer (**B**) and a layer in back (**A**) which contains all the files. As files are created in the TFS directory, they are created in the front layer, and as existing files are modified, they are copied to the front layer. Consequently, the front layer contains all the files that have been created or modified since time 0. In Figure 2b, the file **foo.c** has been created since time 0, while **bar.c** has been modified. When the application decides to take a snapshot of the directory, say at time 1, it creates another layer, **C**, and sets the searchlink in this layer to point to the layer that used to be in front, **B**. From this point on, files will be copied into layer **C** as they are modified. Therefore, layer **B** has become a snapshot of the directory as of time 1. It should be possible to add many layers in this manner.

Since TFS filesystems are made visible by mounting them, one has to mount **B** as the front layer of a TFS directory to see the contents of the directory as of time 1. This is a per-directory mechanism for looking at old versions of files; there is no per-file mechanism like there is in versioned file systems (where, for example, a name like **foo.c[1.2]** could be used to see version 1.2 of the file.)

The TFS was built to make possible the development of the NSE, and one aspect of TFS functionality used by the NSE is the directory revision capability. The NSE's *preserve* command builds new revision layers, as in Figure 2. The NSE's *activate* command performs the TFS mounts needed to make a TFS directory visible. By default, *activate* sets up a view where the newest revision is in front, but one can give a revision name to *activate* to create a view in which an older revision's layer appears in front.

Private Workspaces

The TFS can be used as a way to manage private workspaces for users modifying files in the same directory. For example, suppose that several users wanted to modify files in the directory **A** shown in Figure 2a. Different front layers, like **B**, could be built for each user, all with searchlinks to the same layer in back, **A**. As a result of the TFS's copy-on-write semantics, each user's modifications will be written into his own private front layer, and so **A** will remain unchanged. Therefore each user is isolated from changes by other users. Using the TFS in this manner avoids the need to make a complete copy of the directory for each user who wants to modify files in it. The TFS can be viewed in this context as a mechanism for making a "logical" copy of a directory.

The NSE's *acquire* command uses the TFS to build private workspaces for users. *acquire* creates a directory hierarchy and sets searchlinks from it to point into a common shared directory hierarchy. The user can then *activate* his private workspace to work in it.

The *mount_tfs* Command

Another application of the TFS is as a way to overlay parts of the filesystem. A *mount_tfs* command has been implemented that allows users to mount a directory over another directory such that the contents of the mounted-on directory show through. For example, a user may want to install a local version of a command in a directory over which he has no control, perhaps because he is mounting the directory from an NFS server. If the user wants to install a local copy of **/bin/make**, he would put the local copy of *make* into a directory, say **/tmp/bin**, then *mount_tfs* **/tmp/bin** over **/bin**. After *mount_tfs* constructs the TFS mount, **/bin** will contain the local copy of *make*, which will be in the front layer, along with the rest of the contents of **/bin**, which will show through from the back layer.

Implementation of the TFS

Two of the main goals in implementing the TFS were 1) no new system calls should be needed to initialize TFS filesystems, and 2) existing programs shouldn't need to be modified or re-linked to use the TFS. To make the TFS transparent to user processes, some mechanism was needed to intercept system calls on files.

User-level NFS Server

In the first implementation of the TFS, the mechanism used was that of making the TFS an NFS server [8], [9]. The user-level program which implemented the NFS server was called the *tfsd*. If one assumed that a machine had client-side NFS in its kernel, then one could register the *tfsd* as an NFS server with the kernel. Consequently, TFS files

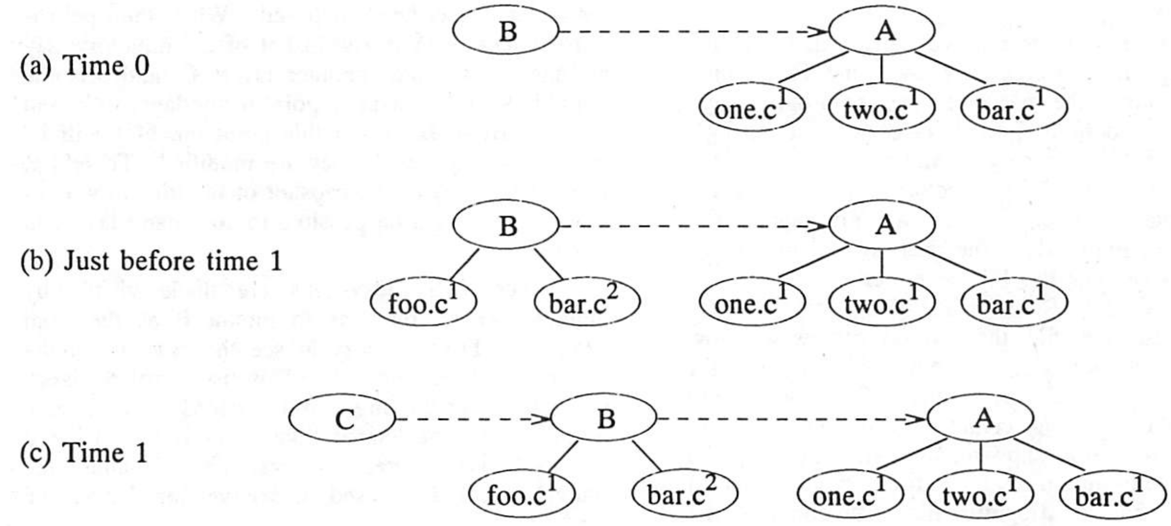


Figure 2: Directory Revisions

would appear as normal NFS files to the kernel, and if system calls were made on these files, the kernel would send NFS protocol requests to the server for the files. The *tfsd* would receive these requests, and perform the appropriate actions on files in the TFS layers. (For example, if the *tfsd* saw an NFS write request for a file which existed in a back layer, it would copy the file to the front layer before performing the write.)

The interface used here to interpose the TFS between the user process and the real filesystem was the interface between the NFS client and NFS server. The advantage of using this interface was that the kernel didn't need to be modified to implement the TFS. The disadvantage was that I/O performance suffered, because all reads and writes on TFS files were translated into NFS read and write requests to the *tfsd*, which became a bottleneck.

Vnode Implementation

Another interface which allows arbitrary filesystem semantics to be interposed between the user process and the real filesystem is the *vnode* architecture in the SunOS kernel [10]. The vnode is the kernel object used to represent a file, and has a number of operations defined on it, such as *open* and *read*. All system calls that operate on files manipulate vnodes by calling vnode operations. The implementation of the vnode operations is filesystem-specific, and is hidden from the code in the kernel that handles system calls. Therefore, to add a new filesystem to the kernel, one only needs to implement the required vnode operations (e.g., *open* and *read*) for the new filesystem, and bind the new filesystem into the kernel.

A set of vnode operations in the SunOS kernel has been implemented for the TFS. The user-level *tfsd* server has been retained, because the TFS functionality implemented in the *tfsd* is too complex and large to put into the kernel.

TFS Name Translation

A TFS vnode in the kernel has a pointer to the "real vnode" for the file or directory, and some TFS vnode operations are translated into operations on the real vnode. This reflects the fact that the TFS performs a name translation between virtual and physical filename, or more precisely, a translation between the filename that the user process sees in the TFS filesystem and the real filename that the TFS sees. Two TFS vnode operations that are translated into operations on a real vnode are the *read* and *write* operations. These operations perform the translation by directly invoking the real vnode's corresponding operation. Because the TFS *read* and *write* operations only add the overhead of one additional function call, the I/O performance of the TFS is virtually the same as that of the underlying filesystem.

The role the *tfsd* plays in this architecture is that of a name server. When a TFS vnode is looked up, a remote call is made to the *tfsd*, which looks up the file and responds with the pathname to the real file. This pathname is then used to look up the real vnode, which is attached to the TFS vnode. So it is the *tfsd* that is responsible for determining the translation between virtual and physical filename. To perform this translation, the *tfsd* follows searchlinks and remembers which layers all the files in a given TFS directory show through from.

The TFS Protocol

All TFS vnode operations that either read or modify a directory must make a remote call to the *tfsd* to obtain their results, so that the *tfsd* can read or modify the appropriate layers of the TFS directory. Consequently, the TFS protocol (the protocol for requests to the *tfsd*), contains requests for both directory read operations (to lookup a file and to read the contents of a directory) and directory write operations (to create/remove/rename a file or directory.) The TFS protocol essentially has the same requests as the NFS protocol, minus the requests in the NFS protocol that operate on individual files, such as *read* and *write*. Therefore, there are two types of TFS vnode operations: operations on files, which call the corresponding vnode operations of the real vnodes, and operations on directories, which make remote calls to the *tfsd*.

As in the NFS protocol, procedures in the TFS protocol have an *fhandle* as one of their parameters. This fhandle is handed out by the *tfsd* to uniquely identify a TFS file, and is opaque to the client of the *tfsd* (that is, the client doesn't look at the contents of the fhandle.) The *tfsd* returns a new fhandle to the client in the results of the *lookup*, *create* (create a file), and *mkdir* (create a directory) requests. When the client gets a new fhandle from one of these requests, it stores this fhandle into the TFS vnode for the file, so that at any time a TFS vnode operation acting on that file knows how to identify the file if it needs to make a remote call to the *tfsd*. Note that the procedures that return an fhandle also return the pathname of the real file, so that the real vnode of the TFS vnode can be determined as described above.

Example of TFS Name Translation

Figure 3 illustrates TFS vnodes and the use of fhandles by the client kernel. Figure 3a shows a tree of vnodes constructed by the kernel. The kernel has one TFS vnode, for the directory **/usr/src**. The other vnodes depicted could be Unix filesystem vnodes or NFS vnodes. The vnode for **/usr/src** has fhandle 1, and its real vnode is the vnode for the directory **/home/jones/test**. If the client tries to find the file **/usr/src/one.c**, it will send a *lookup* request to the *tfsd*, specifying that **one.c** is to be looked up in the directory that has fhandle 1. The *tfsd* responds that **one.c** exists, with fhandle 2 and real pathname **/home/jones/test2/one.c**. The kernel uses this information to create a TFS vnode for **one.c**, resulting in the tree shown in Figure 3b. Notice that the real vnode of **one.c** is in a different directory than the real vnode of **/usr/src**; this means that **one.c** is showing through from a back layer. (The kernel doesn't actually build a tree of vnodes as is depicted here; the trees were drawn to simplify the discussion.)

Comparison of Implementations

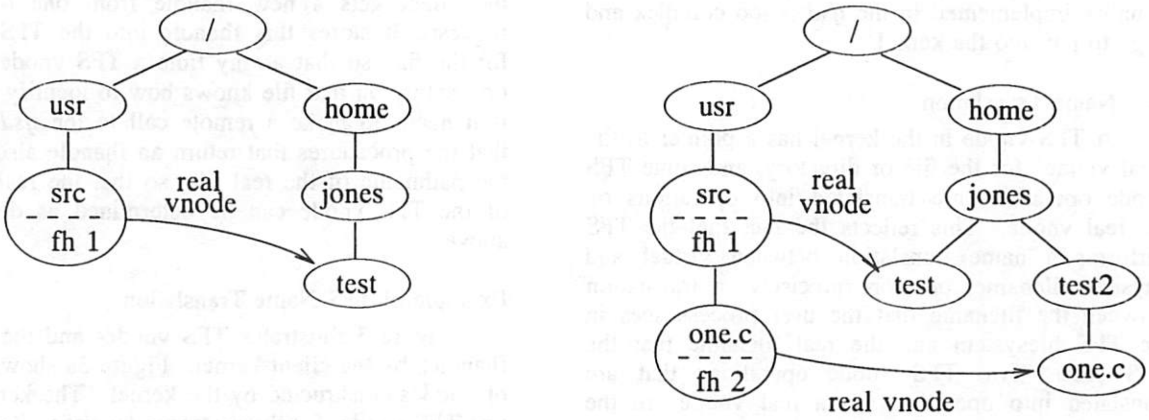
Neither implementation of the TFS requires the addition of new system calls to the kernel. In both cases, the existing *mount* system call is used to initialize a TFS filesystem. Once the mount has been made, standard system calls on files are intercepted and TFS semantics are applied to the file operations.

The vnode implementation of the TFS has better I/O performance than the user-level NFS server implementation, because *read* and *write* requests don't require remote calls to the *tfsd*. In fact, the I/O performance of the vnode implementation is virtually the same as that of the underlying filesystem. However, since TFS vnode operations involve the *tfsd* in all directory requests, the performance of the TFS for commands that do many directory lookups (*ls* or *find*, for example) is the same in both implementations. The drawback of the vnode implementation is that it is less portable than the NFS server implementation. The NFS server implementation is easier to port because the code is all in a user-level server; therefore this implementation doesn't depend on a specific kernel architecture the way the vnode implementation does. Is is also easier to port an NFS server because NFS is a standard protocol that is supported by many operating systems running on a variety of machines.

Internal Structure of the *tfsd*

The internal structure of the *tfsd* is illustrated in Figure 4. When the *tfsd* receives a request from a client, it extracts the fhandle from the request, and looks up the fhandle in a hash table. The hash table entry has a pointer to the *virtual node* for the file. The *tfsd* maintains a virtual node for each file that it sees. The virtual node contains, among other things, a pointer to the *physical node* for the file. The physical nodes are maintained in a tree, and correspond to the files in the real filesystem. The *tfsd* builds pointers between the physical nodes which correspond to searchlinks it has followed. The physical nodes are used to determine the real pathnames that should be passed back to the client, and that should be used by the *tfsd* when it performs directory operations.

Figure 4 shows the internal state of a *tfsd* that is serving a TFS filesystem with root directory **/home/jones/test**. The root directory has a searchlink to **/home/jones/test2**, which is represented by the dotted line from the physical node for **/home/jones/test** to the physical node for **/home/jones/test2**. The files in the root directory, **one.c** and **two.c**, are showing through from two layers. The *tfsd* has assigned fhandles 1, 2, and 3 to the root directory, **one.c**, and **two.c**, respectively. Note that the *tfsd* picks an arbitrary name for the root virtual node; it does not know the client's name for this directory. If a client had mounted this TFS filesystem onto **/usr/src**, then Figure 3 shows the vnodes that the client would build when viewing this



(a)
Before lookup

(b)
After *lookup*(**one.c**, fhandle = 1)
call to *tfsd*

Figure 3: TFS Vnodes in the kernel

filesystem. In Figure 3, the client sent a *lookup* request for **one.c** in the directory with fhandle 1. One can see in Figure 4 what the *tfsd* does with this request. It finds fhandle 1 in the fhandle hash table, and follows the pointer to the virtual node for **root**. It then discovers that one of the child nodes of **root** is a node for **one.c**, with fhandle 2. Looking at the physical node for **one.c**, it finds that the real name of the file is **/home/jones/test2/one.c**, and returns this name to the client, along with the fhandle 2.

Future Work

Per-File Sharing

Currently files are shared on a per-directory basis. This creates a problem if a version control tool would like to create a private workspace for a user in which some of the files are shared from a layer **A** and the others from a layer **B**, where **A** and **B** aren't connected by searchlinks. In such a case, the new front layer can only have a searchlink to either **A** or **B**, and the files to be shared from the other layer must be copied into the new front layer. The problem here is that the granularity of sharing is too large; it should be on a per-file basis.

Per-file sharing can be effected by eliminating the searchlinks to read-only layers and instead using a file which has pointers to individual files in read-only layers. In effect, this file would contain per-file searchlinks.

Making the *tfsd* Stateless

The *tfsd* is a *statefull* server, that is, it uses past state to answer current requests. This means that if the *tfsd* dies for some reason, it is difficult for the client to recover. Even if a new *tfsd* starts up, it will not have the state necessary to answer the client's requests. If the *tfsd* were stateless, crash recovery would be greatly simplified, and crashes could be hidden from the user. Unfortunately, it is not easy to make the *tfsd* stateless without sacrificing performance.

The main reason the *tfsd* is statefull is that it remembers all the fhandles it assigns to files, but doesn't save them to persistent storage. Consequently, if the *tfsd* dies for some reason, its knowledge of its fhandles goes with it, and the client is left with invalid fhandles. Making the *tfsd* stateless requires that it write all of its fhandles to persistent storage, so that if it dies, another instantiation of the *tfsd* can start up and serve the same set of fhandles. This means that whenever a new fhandle is assigned, the fhandle would have to be written out before any client is given it. More research needs to be done to determine if this can be done without sacrificing too much performance.

Conclusion

The Translucent File Service can be used to make "logical" copies of files, rather than physical ones. This is useful for 1) making snapshots of directories in order to save the history of some set of objects, and 2) allowing developers to work in parallel on a set of sources without physically copying the sources. Two implementations of the TFS have been done: as a user-level NFS server, and as a set of vnode operations in the SunOS kernel which dispatch out to a user process, the *tfsd*. The performance of the vnode implementation is better than that of the user-level NFS server implementation. The NFS server implementation of the TFS was completed in the summer of 1987, and was bundled as part of the initial release of NSE in early 1988. The vnode implementation of the TFS was implemented during the summer of 1988, but was not sent to customers until NSE version 1.2 was released in

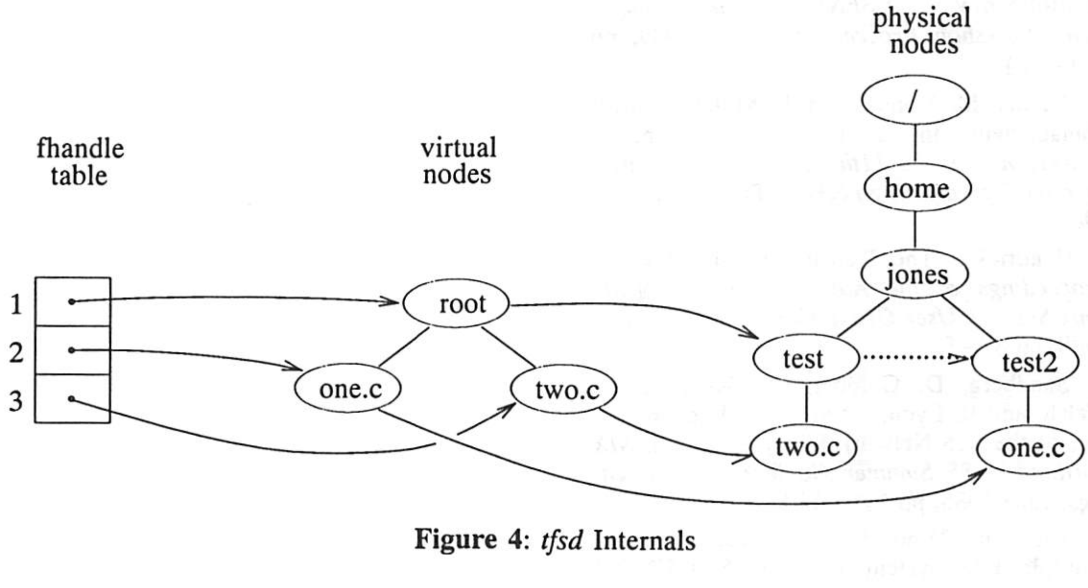


Figure 4: *tfsd* Internals

the fall of 1989. The vnode implementation of the TFS is also a part of SunOS version 4.1.

The TFS has proved to be extremely useful for the implementation of Sun's version configuration and management tool, the Network Software Environment (NSE). The TFS is a general enough service that there are undoubtedly other interesting applications that could make use of it.

Acknowledgements

Many people at Sun helped design the TFS. Here is a fairly complete list: Evan Adams, Azad Bolour, Rob Gingell, Tom Lyon, Terry Miller, Marty Honda, and Russel Sandberg.

References

[1] [1] V. B. Erickson and J. F. Pellegrin, "Build — A Software Construction Tool," *AT&T Bell Laboratories Technical Journal Vol. 63 No. 6 Part 2*, July-August 1984, pp 1049 - 1059.

[2] G. S. Fowler, "The Fourth Generation Make," *USENIX Portland 1985 Summer Conference Proceedings*, June 1985, pp 159 - 174.

[3] R. Brender, "Generation of BLISSes," *IEEE Transactions on Software Engineering*, November 1980, pp 553 - 563.

[4] D. Korn and E. Krell, "The 3-D File System," *USENIX Baltimore 1989 Summer Conference Proceedings*, June 1989, pp 147-156.

[5] D. Leblang, R. Chase, Jr., and G. McLean, Jr., "The DOMAIN Software Engineering Environment for Large-Scale Software Development Efforts," *Proceedings of the 1st International Conference on Computer Workstations*, IEEE Computer Society, San Jose, CA, November 1985, pp 226-280.

[6] A. Hume, "The Use of a Time Machine to Control Software," *USENIX Software Management Workshop Proceedings*, April 1989, pp 119 - 124.

[7] E. Adams, M. Honda, and T. Miller, "Object Management in a CASE Environment," *Proceedings of the 11th International Conference on Software Engineering*, Pittsburgh, May 1989.

[8] D. Hendricks, "The Translucent File Service," *Proceedings of the Autumn 1988 European Unix Systems User Group Conference*, October 1988, pp 87 - 93.

[9] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, "Design and Implementation of the Sun Network Filesystem," *USENIX Portland 1985 Summer Conference Proceedings*, June 1985, pp 119 - 130.

[10] S. Kleiman, "Vnodes: An Architecture for Multiple File System Types in Sun UNIX," *USENIX Atlanta 1986 Summer Conference Proceedings*, June 1986, pp 238 - 247.

David Hendricks is a member of the technical staff of Sun Microsystems, and is a member of the Network Software Environment group. His interests include Computer-Aided Software Engineering, networking software, and exotic filesystems like the TFS. Dave received an MS in Computer Science from Stanford University.

THE USENIX ASSOCIATION

The USENIX Association is a not-for-profit organization of those interested in UNIX and UNIX-like systems. It is dedicated to fostering and communicating the development of research and technological information and ideas pertaining to advanced computing systems, to the monitoring and encouragement of continuing innovation in advanced computing environments, and to the provision of a forum where technical issues are aired and critical thought exercised so that its members can remain current and vital.

To these ends, the Association conducts large semi-annual technical conferences and sponsors workshops concerned with varied special-interest topics; publishes proceedings of those meetings; publishes a bimonthly newsletter *;login:*; produces a quarterly technical journal, *Computing Systems*; serves as coordinator of an exchange of software; and distributes 4.3BSD manuals and 2.10BSD tapes. The Association also actively participates in and reports on the activities of various ANSI, IEEE and ISO standards efforts. Most recently, the Association created UUNET Communications Services, Inc., as a separate not-for-profit organization offering electronic communications services to those wishing to participate in the UNIX milieu.

Computing Systems, published quarterly in conjunction with the University of California Press, is a refereed scholarly journal devoted to chronicling the development of advanced computing systems. It uses an aggressive review cycle providing authors with the opportunity to publish new results quickly, usually within six months of submission.

The USENIX Association intends to continue these and other projects, and in addition, the Association will focus new energies on expanding the Association's activities in the areas of outreach to universities and students, improving the technical community's visibility and stature in the computing world, and continuing to improve its conferences and workshops.

The Association was formed in 1975 and incorporated in 1980 to meet the needs of the UNIX technical community. It is governed by a Board of Directors elected biennially.

There are four classes of membership in the Association, differentiated primarily by the fees paid and services provided.

For further information about membership, write or call

USENIX Association
Suite 215
2560 Ninth Street
Berkeley, CA 94710
phone: +1 415 528-8649
e-mail: uunet!usenix!office or office@usenix.org

USENIX SUPPORTING MEMBERS

AT&T Information Systems
Open Systems Foundation
Convex Computer Corporation
Digital Equipment Corporation
Quality Micro Systems
mt Xinu
Sun Microsystems, Inc.
Sybase, Inc.
The Aerospace Corporation